Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

CHARLES UNIVERSITY PRAGUE

faculty of mathematics and physics

Serghei Bondari

Distributed Web Framework

Department of Software Engineering
Supervisor: RNDr. Filip Zavoral, Ph.D.

Study program: Computer science, software systems

2009

I would like to thank David Borgesen, from Aspiresoft corp. for the opportunities he gave me and for the help he provided during my work on this thesis.

I would also like to thank Danel Sevcik, from Aspiresoft corp. for reading preliminary versions of the thesis as well as giving me suggestions and feedbacks.

I would also like to particularly recognize and thank my wife Olga and my daughter Maria for their constant support of my endeavors, whatever they have been.

Finally, I thank my supervisor, RNDr. Filip Zavoral, Ph.D. for not being satisfied with some of my initial work, guiding me and encouraging me to make it better.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 15.04.2009 Serghei Bondari

Název práce: Distributed web framework
Autor: Serghei Bondari
Katedra (ústav): Katedra softwarového inženýrství MFF UK
Vedoucí diplomové práce: RNDr. Filip Zavoral, Ph.D.
e-mail vedoucího: zavoral@ksi.mff.cuni.cz
Abstrakt: Navrhnout a vyvinout distribuované prostředí pro vývoj distribuovaných komponentových web aplikací v jazyce PHP5. Prostředí umožňuje vývoj nezávislých znovuvyužitelných aplikačních komponent. Součástí práce je definice rozhraní komponent a implementace vrstvy middleware, která umožňuje posílání událostí a dat mezi komponentami.
Klíčová slova: PHP, distribuovaný system, prostředí

Title: Distributed web framework
Author: Serghei Bondari
Department: Department of software engineering, faculty of mathematics and physics.
Supervisor: RNDr. Filip Zavoral, Ph.D.
Supervisor's e-mail address: zavoral@ksi.mff.cuni.cz
Abstract: Develop a distributed component based web framework built upon the PHP5 programming language that allows independent component development, deployment and active component re-usage. A component of the work will recommend and implement a component interface definition schema and effective middleware layer that facilitates events, messaging and data exchange between various components.
Keywords: PHP, framework, distributed

Contents

List of Tables

List of Figures

Chapter 1. **Introduction**

1.1. Motivation

In a relatively short period of time, the PHP programming language has become a web development industry standard. Its ease of use and robust functionality has proven to be well suited for not only for simple projects, but also for complex application development platforms in many mission critical web applications. In version 5, PHP has obtained an advanced object model of which is extremely similar to the object models in C# and Java programming languages. This robust object model allows for the creation of large and sophisticated object oriented frameworks written in PHP.

With each page load, the PHP code must be loaded and parsed, as well, it needs to run the initialization code in order to preserve the execution state between page processing sessions. As PHP makes its way into the enterprise, these aforementioned limitations create new technical challenges relating to performance and scalability.

A big advantage of having a distributed component based web application is that load, caused by a request processing may be spread amongst several servers. By doing this we are spreading the load, thus avoiding complex and expensive clustering solutions where it can be avoided.

These problems and issues can be resolved by a PHP application framework, a system that would allow the definition of applications and execution of these applications in a distributed environment.

1.2. Goals

The goal of this work is to create an application framework and a methodology for defining and writing distributable web applications that will allow parallel server and distributed execution of application code on multiple system nodes. This project is challenging due to the fact that PHP lacks support for creating and running multiple execution threads.

The system will need to be designed to implement distributed objects invocations, remote exceptions and various other features that need to provide a high level of system transparency that hides the fact that the application is actually running in a distributed environment. Another core function of the framework will be a distributed document generation and processing process that mechanism to build and process documents by many components on multiple nodes in parallel, so that project instance will behave as an application cluster to

share the computational load across multiple computers. In summary, the main goals of the project are:

- Design and implement a distributed objects functionality
- Develop a distributed document processing system
- Develop a component model and definition schema that provides a high level of component reusability, independence and autonomy
- Design and implement a distributable application platform to provide the necessary infrastructure for all the items above

1.3. Project Name

This project shall be referred to as the MOSAIC application framework.

1.4. License

The MOSAIC application framework is an open-source project, distributed under the BSD license.

1.5. Obtaining the Source Code

The source code of the project is available on at https://dev.bondari.net/trac/mosaic and is contained in the subversion repository. The recommended way of obtaining the most recent development snapshot is to export it from the SVN repository located at svn://dev.bondari.net/mosaic/trunk.

Chapter 2. **Contemporary Solutions**

2.1. Enterprise Java Beans (EJB)

The closest technology utilizing a component model similar to the goals of the MOSAIC project is the implementation of Enterprise Java Beans. The goals of MOASIC are very similar to the implementation of EJB of which is very mature, robust and highly advanced.

EJB consists of 3 main parts:

- EJB Component
- EJB Container
- EJB Object

EJB component is a Java class that implements business logic. The EJB Components' code is executed inside the EJB Container. The EJB Container provides and implements many services used by EJB Components, such as transactions, resource management, versioning, and lifecycle. One EJB Container usually contains many similar EJB Components.

EJB Objects are objects that allow client applications to call methods on remote EJB Components, acting as a middleware between the client and EJB Component.

2.2. Java Remote Method Invocation (RMI)

Java RMI is a distributed object model that specifies remote execution of an object's methods on a distant Java virtual machine. When an application wants to call a remote object's method, it is calling it on the local object instance called a *stub*. The Stub object encodes parameters to be platform independent (for example, integers are always encoded as big-endian) and parameters are serialized into binary format. This process is called *parameter marshalling*. The main goal of marshalling is to convert method call parameters into a format suitable for transferring them through the network. Each block of data also contains identification of the remote object and a description of the called method. Stub sends this block to the target JMV.

A recipient object, called a skeleton, performs a parameter *unmarshalling* (a process opposite to marshalling, when parameters are being converted back into their original types), calls a method with the parameters and then marshals return value and sends it back to stub.

2.3. CORBA

CORBA is a rich, extensive family of standards and interfaces that is considered an industry standard. Compared to RMI, RMI is easier to use than CORBA. The CORBA Services specification includes comprehensive high-level interfaces for naming, security, and

transaction services. Another substantial difference between RMI and the CORBA specification is that RMI tends to be more a client-server model, where CORBA is a peer-to-peer ORB communication model. Although some of these interfaces have some interesting approaches, delving into the details of these interfaces would be overkill for the task at hand.

Unlike the JAVA technologies described above, the Common Object Request Broker Architecture (CORBA) is not so much a distributed system implementation, but a very advanced and detailed specification of it. This specification was laid down by the Object Management Group (OMG) that was formed in 1989 to develop, adopt and promote standards for the development and deployment of applications in a distributed heterogeneous environment [19]. The main aspects targeted in CORBA specification are:

- ORB Core
- OMG Interface Definition Language
- Interface Repository
- Language Mappings
- Stubs and Skeletons
- Dynamic Invocation and Dispatch
- Object Adapters
- Inter ORB protocols

2.3.1. ORB Core (ORB)

The primary goal of the ORB[1] is to deliver a request to the object and return a response to the client. A key feature of ORB is to provide multiple types of transparencies:

Object location transparency hides the actual location of the object from the client. It may be on the same machine or same process or even on other machine over the network.

Object implementation transparency hides from the client the actual programming or scripting language that was used to write the object.

Execution state transparency allows a client to make requests on the object that may not be active (executing state). In such situations ORB may start the object in order to deliver the request.

Communication mechanism transparency abstracts is a way for the client to communicate with the object via TCP/IP, IPC[2] or just a function call.

[1] Object Request Broker
[2] Inter Process Communication

ORB also defines a term *object reference* that brings all necessary information about how an object can be reached.

2.3.2. OMG Interface Definition Language (ILD)

This is a language independent way to describe to the interface what methods and types objects are supported, thus defining the requests an object can handle. IDL also provides mapping to the real underlying programming languages. Despite the simplicity of such data types as long, float, char etc., the CORBA specification allows constructed types, which are unions and structures. IDL also supports exception descriptions and interface inheritance. Being as simple as possible, OMG IDL may be used with almost any programming language.

2.3.3. Stubs and Skeletons

The OMG IDL language translator is also able to generate stub client-side and server-side skeleton code. Stub is responsible for issuing the request on behalf of the client and skeleton delivers the request to the CORBA object implementation. When using a stub/skeleton invocation strategy, we refer to this as *static invocation*.

2.3.4. Dynamic Invocation

CORBA defines a Dynamic Invocation Interface (DII) for client side, and Dynamic Skeleton Interface (DSI) for servers. Dynamic invocation basically can be referred to as a generic stub and generic skeleton strategy. Unlike the stub/skeleton approach, it is not dependent on IDL.

Chapter 3. **Analysis**

3.1. Distributed systems

3.1.1. Definition

There are many interpretations of the term distributed system; for this project, we will define it as follows [1]:

> *"A distributed system is a collection of independent computers that appears to its users as a coherent system."*

This definition covers two essential aspects of distributed systems. The first aspect demands single machine autonomy and the second encompasses the software, users, and where high level applications think they are running on a single system. Most distributed systems allow these abstractions by using a layer in-between applications and operating systems, called *middleware*. This middleware layer is responsible for creating a single system environment above multiple systems through the network.

The theory of distributed systems defines several goals to be accomplished: transparency, openness and scalability. Here we will review which aspects and goals are to be achieved in MOSAIC system.

In addition to the term "distributed system", there are several terms that will be used in this document repeatedly that need to be defined:

***Node**: An individual computer that runs the system*

***Site**[3]**:** A collection of nodes*

***Agent:** An independent subsystem or component of the site performing specific function*

[3] In literature "site" is usually referred to the subset of nodes. We will use "site" as the set of all nodes of the system, and "sub-site" to indicate that we are talking about partial collection of nodes.

3.1.2. Transparency forms

There are different of forms of transparency in a distributed system.

Transparency	Description
Access	Hide the differences in data representation and how resource is accessed
Location	Hide resource location
Migration	Hide that resource has moved to another location
Relocation	Hide that resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource is being used by several competitive users
Failure	Hide the failure and recovery of a resource

Table 1. Forms of transparency in distributed systems. (ISO, 1995)

Access transparency describes communication over the network where the application is unaware of the fact that it is accessing data through the network.

Location transparency refers to the fact that a user cannot tell where the resource is located. In MOSAIC location transparency combined with *migration transparency* will be one of the key factors to provide a high level of component autonomy and independence.

Replication is a form of transparency where *data* or a *replicated service* appears identical on different nodes of the system. A request to the replicated resource may be directed to any of the nodes. Although replication will not be implemented, the MOSAIC project architecture can be enhanced to provide limited resource replication. Replication is a key factor in providing scalability and reliability of a MOSAIC based site.

Concurrency transparency, *relocation* and *failure recovery of resources* are beyond the scope of this project's goals and are also greatly limited by the capabilities of PHP.

3.2. Middleware

Definition of a distributed system implies that several computers with an operating system installed and connected through the network are not meeting criteria of coherence. To accomplish this coherence, a new layer of software is therefore introduced in most of the modern distributed systems applications.

3.2.1. Middleware model

Each middleware implementation is based on a model, sometimes referred as a middleware *paradigm*. According to [1], several middleware paradigms appeared during the evolution of distributed systems:

- distributed file system model
- RPC[4] based middleware
- distributed objects
- distributed document model
- distributed event based system (DEBS)

A distributed file system model is most suitable to build a distributed operating system that follows the UNIX approach, where nearly every resource is a file. Distributing the file system would be a natural choice for such systems.

RPC is a very popular paradigm for implementing client-server communications. This paradigm is well suited for many implementations and is easily understood by developers. However RPC from its nature usually implies a set of substantial restrictions on communications. RPC enforces tight integration of the communicating parties, often forcing to the parties to perform synchronous communication or remote resource polling of which may greatly influence load, scalability and implementation of client and server components.

Unlike in the RPC middleware model, the Distributed Event Based System (DEBS) paradigm, publishers[5] are unaware of what component will receive a notification, and respectively subscribers[6] are not interested in what component issued an event, and are more concerned about the event itself. Such an architectural approach allows the component design to be very autonomous, reusable and independent, as the event-based components are not designed to work with specific components. This facilitates the effective separation of business logic and communication, thus bringing in to the distributed system design a very advanced level of access, location, migration and replication transparencies simultaneously.

The DEBS paradigm also brings another benefit to the project in comparison to RPC. The RPC Interface Definition Languages (IDL) usually tend to be quite complex and type-strict, of which is not necessary in PHP and it breaks the "KISS" philosophy of keeping it simple in PHP. Instead of RPC IDL, in our distributed event-based system we shall use a component

[4] Remote Procedure Call
[5] A component that issues an event notification
[6] A component that manifests its interest in receiving of an event notification

subscription[7], which is unlike IDL and easier to describe. The MOSAIC project uses the subject-based subscription mechanism because of its simplicity and ease of implementation. This method is powerful enough to accommodate all necessary facilities for Inter Component Communication and distributed documents processing. Each event notification type has the unique string identification, where a certain level of abstraction may be taken as a RPC procedure name, or, in the case of the distributed document processing, the event name represents a named part of the document to be processed by the component. With this solution both inter component communication and distributed document processing related communications are implemented in a very similar manner.

Due to the simplicity of a subject-based subscription mechanism, it may not be suitable for extremely large and/or geographically scattered systems. In general, a distributed event-based system middleware paradigm allows the MOSAIC system components be more simple, independent, self-focused, reusable and autonomous, all of which exactly matches our goals. Further reading about DEBS is available in [2].

In order to provide a distributed document processing subsystem, the MOSAIC project middleware also implements a distributed document middleware model. The most popular real world example of this paradigm is the World Wide Web, of which is organized into the documents residing on millions of servers (nodes), linked with hyperlinks. MOSAIC exploits distributed document paradigm in a bit different manner than the World Wide Web. Unlike in WWW where complete documents are stored on different nodes, MOSAIC assembles parts of a single document called *tiles* of which are distributed over the system. These tiles are recursively asynchronously processed in parallel by distributed application components.

To summarize, the MOSAIC project middleware follows 3 middleware paradigms:

- distributed event based system model
- distributed object model
- distributed document model

3.2.2. Middleware facilities

From the nature of distributed systems all middleware systems must implement access transparency, of which mostly comprises of the *network communication facilities, naming facility, persistence*, *distributed transaction* and *security facilities*.

[7] Subscription describes a set of notifications a subscriber component is willing to receive.

Persistence and transaction facilities are related to the distributed data storage functionalities; security facilities introduce access rights to resources of a distributed system. None of these services are required for accomplishing project's goals so they may be left for future extensions of the project.

Communication facility

The implementation of these facilities greatly depends on the middleware paradigms employed, system goals and demands but also lower-level capabilities of the under laying operating system and network.

We shall not give much attention to the each layer of the OSI[8] [3] model, as it is not necessary for our goals. However we need to discuss 2 main layers which are directly used in the MOSAIC project.

First we will shortly discuss the selection of the transport layer protocol, where in a typical modern network environment can be either TCP [4] or UDP [5]. TCP is a reliable connection-oriented protocol, but has an overhead drawback comparing to the connectionless UDP. On the local network their reliability may be treated as equals and UDP might look as a better choice for a message exchange system. On the other hand UDP would more likely require the introduction of additional flow and error functionality, optimized for the particular application.

The MOSAIC project will use the TCP protocol for transport, but with an additional design benefit to reduce overhead while establishing and closing the TCP connection. A higher level middleware protocol will implement multiplexing to pass data through the same opened TCP socket requests and responses, thus reducing the TCP flow control communication to a minimum. This approach will provide all the advantages of the TCP communication, such as connection, reliability and error detection, while minimizing overhead related to creating or closing a connection.

The communication layer I want to shortly stop attention on is the middleware protocol which covers a presentation and an application layer of OSI model. As I have mentioned above, our middleware protocol must implement a message multiplexing control. In addition to this functionality, the protocol must be able to transfer event notifications, remote object invocations data and also accommodate naming service data and control messages. Middleware protocols sometimes implement another useful feature as the multicasting support; however multicasting support is not required by our goals and limited multicasting functionality is implemented on the application level, where we and the protocol to be simple.

[8] Open System Interconnection

Naming facility

Main goal of the middleware naming facility is to provide location and migration transparencies, especially in terms of the distributed objects implementation and resources location and migration. Though these goals may be accomplished without a naming facility just by adding network connection information into the remote objects references and resources, such a solution would have several major drawbacks.

First, if the network address of the server changes due to some failure or other reason, all object references would become invalid. Second, a remote object reference will also have to contain information where to get stub code that client may load during the binding procedure. In addition, such approach obviously lacks any system scalability and reliability awareness.

For these reasons the MOSAIC system does have a naming agent that is a "well known"[9] service in the MOSAIC site, thus any node can query the remote object or other node resources actual location on the network. When going online, each node is obliged to register its remote objects, resources and its network location with the naming service.

The MOSAIC naming facility functionality is also extended to be a site *control agent* for the simple reason of its awareness of the current site configuration based on its definition, thus making it a suitable candidate for this position. The control agent is simply another "well known" service that is responsible for maintaining information about the site configuration, monitoring node status and propagating nodes status change notifications to the remaining nodes. Though in some distributed system applications this agent may be a separate and independent service, in the MOSAIC project the naming agent is perfectly capable of performing this role, thus reducing the implementation coding volume and site complexity.

3.2.3. Inter Component Communication

The Inter Component Communication (ICC) middleware subsystem, as the name implies, allows system components to communicate to each other. As discussed in the "Middleware model", instead of a much more common and conventional RPC model, the MOSAIC project uses a distributed event-based system paradigm (DEBS).

The MOSAIC event notification service is not only used for a one-way subscriber notification about the change of state of publisher, but also as a generic way to request for data from event subscribers, replacing conventional RPC calls. With this we enhance the RPC model, where the server end-point is well known to the caller, with powerful separation of publishers and subscribers in DEBS. Also, the calling component is unaware of the callee

[9] In our terms, "well known" means that network address of this service is defined with the node configuration, so that no other information is required to connect to the "well known" agent.

component, relying on the middleware layer to select and deliver the request to the correct component instance. Another change that is brought by this mechanism unlike in RPC, there can be several recipients and processors of the event notification. This also means that by issuing an event notification, the client may even gather data from more than one server component.

The event notification returns to the publisher after being delivered to all its subscribers. Each subscriber of such notification has capability of adding response data in to the event object that is floating across the distributed site. The object with a notification is then delivered back to the component that issued a notification. This process is similar to a boomerang and is referred to as *boomerang events*. Main principle of such events is illustrated on the Figure 1.

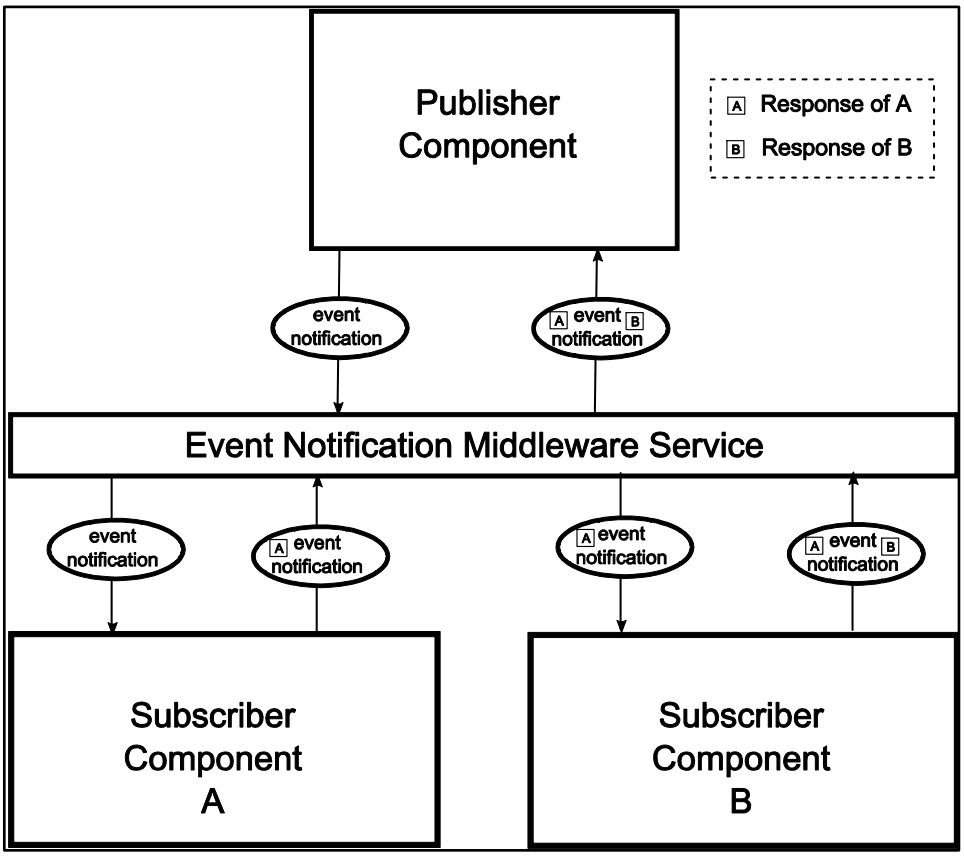

Figure 1. MOSAIC boomerang event notification service

Due to the importance of both communication paradigms the MOSAIC system implements both *synchronous* and *asynchronous* methods when issuing an event notification.

In the case where a caller needs response data later, it must define a callback function[10] that will be called at the appropriate time without the execution interruption[11]. Until that time,

[10] PHP 5.3 introduced rich lambda function implementation, called closure. Implementation of such callbacks would be impossible in the earlier version.

the response is held in the middleware request pools. Without a callback defined, asynchronous event notification behaves like a response is not required by the client and, as such, the response is omitted.

Synchronous event notification is done in accordance with the *deferred synchronous RPC* execution strategy explained in [1]. In RPC one synchronous call is replaced with 2 asynchronous calls. When a caller component issues a synchronous boomerang event, its execution is suspended. While waiting for a synchronous response, a node may execute another component's code. After the boomerang event returns, execution of a component may continue with the data from a response in hand. With some certain level of abstraction, this approach is used in OS kernels during the *syscall* operation, when the process that invoked the syscall is suspended and other ready process is scheduled to run until resources or data demanded by the syscall are ready. The MOSAIC based node simulates the behavior of a *non-preemptive* OS kernel. More information on this topic will be given in the implementation part of this document.

3.2.4. Parameters Passing

Choosing a parameters passing strategy solution is a common and very complex problem of ICC and distributed objects subsystems. The solution greatly influences future capabilities and limitations of the framework. There are two basic ways how parameters can be passed: it's either by value or by reference.

Passing a parameter by value is the simpler option and requires only a suitable serialization mechanism to convert parameter data into the presentation suitable for transfer through the network. Unlike many programming languages, PHP provides a native built-in serialization mechanism that, despite of simple scalar types like integers, floats, strings and Booleans, allows serialization of complex types like arrays and objects. The only limit of the built-in PHP serialization mechanism is that it is unable to serialize language supported type "R*esource"*. Mostly, it does not make sense to pass a resource to the remote node in MOSAIC as the resource is usually a local system resource representation such as a file descriptor or network connection and thus are not passable to the remote system. However the MOSAIC distributed document processing mechanism requires serialization/deserialization of a DOM[12] type resource. So in order to simplify the implementation of distributed a XML based

[11] Implementing interrupts in PHP stops at limited capabilities of environment where the only possible way to interrupt PHP execution thread is using POSIX signals; without multithreading support it does not make much sense to do it.

[12] Document Object Model – object model representing a XML document

documents processing mechanism, the PHP serialization mechanism is extended to support serialization and de-serialization of a DOM resource type.

Passing a parameter by reference turns out to be a more complex task because of the fact that the actual data modification is being done on the remote computer, and every change that is done remotely must be reflected locally. There are several possible ways how to solve passing parameters by value. In the case of simple structures and arrays, parameters are copied back to the callee after caller has modified it. With this approach it is obvious that network communication may be significantly increased and such an approach cannot cover arbitrary data structure transfers. This approach also makes impossible concurrent access to the referenced data.

For these reasons, the MOSAIC project uses the solution that does not pass actual data for modification, but generating a special code for using the data remotely – distributed objects. Instead of passing an object by reference, we just pass a serialized stub instance by value. Such an approach naturally works as a passing an object by reference.

3.3. Distributing Objects

From the nature of object-oriented programming, principles of RPC can be equally well applied to objects. Here we will expand upon the RPC implementation to accommodate the idea of remote object invocation, and enhancing the distribution transparency level compared to RPC. There are several fundamental design problems that should be discussed before doing the actual remote object implementations.

3.3.1. Compile time and Runtime Objects

Objects in a distributed system may appear in several forms. The most common form is when the object is represented by the object oriented language facility, such as *class*. A remote object then is defined as an instance of a class. Using *compile time objects* is natural and makes it very easy for developer to work with such objects. A disadvantage of such approach is that such objects are strictly bound to the programming language.

A *runtime object* allows the mixing of objects written in different languages in one application. Such an object may actually be a dynamic C library with some state whose functions are called in the distributed object manner. To unify such an approach the *object adapter* is created above such entities to give it an object appearance.

Due to the fact that the MOSAIC project is implemented in PHP it has a rich object model, using a compile time objects style is a reasonable choice. This fully covers all the requirements of the project and the implementation of an object adapter is unnecessary.

3.3.2. Binding

Binding is the operation when the stub is being created in the client's address space in order to start invoking the remote object. In speaking about *implicit binding* there are no special methods or functions that must be called in order to start making remote calls. As opposed to implicit binding, *explicit binding* requires a special binding call in order to link the object reference to the remote object. PHP version 5 provides a powerful infrastructure to implement completely automatic and transparent implicit binding, such as on-demand class code loading, method call interception etc. Utilization of these features allows for complete transparent usage of remote objects in the code, thus implementing the implicit binding policy.

3.3.3. Object references

A distributed object reference must provide enough information to allow a stub to bind to the remote object. Minimal information should contain a network machine address and some unique object identification on the server. This straightforward approach has several drawbacks discussed in the "Middleware facilities: lack of naming facility drawbacks" analysis. In addition to these issues, the future project extension we should not assume that the client and server would use the only agreed protocol; therefore, the MOSAIC project also adds a communication protocol identifier into the object reference although it is unnecessary at this time.

To summarize, object reference in MOSAIC includes a communication protocol, the name of the node that accommodates object instance and the unique object identification on the server. By following WWW standards and conventions, the string identifier of the remote object will be later referred as object URI and have a following format:

```
<protocol>://<node_name>/<object_identifier>
```

3.3.4. Garbage collection

Because of the fact that the MOSAIC projects goal is to implement a persistent application server, we should pay additional attention to the one of the most important aspects of the memory management area of which is the de-allocation of unused memory blocks.

Removal of objects that are no longer needed can be generally done in 2 ways. The first approach is the explicit or manual removal. Although this approach is very clean and straightforward, such an approach imposes many difficulties within distributed systems. It is typically unknown whether the object is referenced and intended to be used from somewhere in the site.

So the removal of such object is done automatically, when it is "no longer needed" or in other words, it is *unreferenced*. An unreferenced entity can safely be removed from the distributed system site memory. This automated facility is called the *garbage collector*.

Reference Counting

In non-distributed systems garbage collection is often implemented via reference counting. Most of the advantages and drawbacks referring to counting strategies are well described in [1] , [6] and [7]. In a distributed system this method is greatly affected by the reliability of communication. A message with referencing or dereferencing manifest may be lost or retransmitted, leading to an invalid reference counting value.

In addition to the network reliability dependence, a very simple race condition illustrated in [1] may happen resulting in early object dereferencing. When process P_1 passes a reference to the process P_2 and immediately decides to dereference the object the skeleton receives a dereferencing message before a reference actually reaches P_2, and the reference count becomes zero.

To resolve this, P_1 needs to pass the reference increment intention before passing the actual reference to P_2. Then process P_2 needs to acknowledge that it actually received a reference. This strategy is overcomplicated and ineffective.

To reduce load on the network and to address issue above, MOSAIC uses very efficient and simple Generation Reference Counting (GRC) algorithm introduced in [6].

Generation Reference Counting

Each stub contains its *generation index* G(S) and *copy counter* C(S), where S denotes a stub. When the first reference is created, the stub has both numbers set to zero, as if it was never copied and it's said to belong to generation 0.

When we copy stub S_1 to S_2 we are sending S_2 to the remote process in a common way. However C(S_1) = C(S_1) + 1 because it was copied. C(S_2)=0 because it was never copied. In addition to this, S_2 is said to belong to the *next generation*, thus G(S_2) = G(S_1) + 1. No message is sent to skeleton at this point.

Every skeleton contains a vector *GRT*[13], where *GRT*(i) is the number of references of the generation *i*. When stub S is removed, a message containing the G(S) and C(S) is sent to the skeleton. Skeleton updates its GRT vector as following:

GRT(G(S)) = *GRT*(G(S)) – 1

GRT(G(S)+1) = *GRT*(G(S)+1) + C(S)

[13] Generation Reference Table

In other words, one stub of generation G(S) is removed, but this stub was copied C(S) times to the stubs of the next generation. When $GRT = \vec{0}$ object is no longer referenced and may be removed.

3.4. Component Definition Schema

The Component Definition Schema is a document that describes a component interface and resources provided or required by a component. Some of the frameworks written in PHP use PHP itself to describe a module or package contents. By including this file, the PHP code is executed and the package configuration is loaded into the previously agreed global variable. This approach is quite problematic and counterproductive for the distributed multi-node system where such manifests must be somehow transferred between system nodes; they also might be united with other manifests or even loaded into different configuration variable that are initially supposed to be loaded.

Another approach would be to use the proprietary XML schema to describe the package contents and resources. This is a much better option compared to previous one, but it has several disadvantages relating to the final solution used in MOSAIC project. This project shall use the Resource Description Framework (RDF) for the Component Definition Schema.

Using RDF brings many advantages, especially from the semantics and implementation techniques. In the first row, the RDF schema supports many constructions that could be used for component definition out-of-the-box such as: creator, date, titles and other metadata fields described in Dublin Core Metadata Initiative (DCMI) [8]. Other useful elements of RDF include sequences and bags for ordered and unordered sets, data types support, and easy extensibility of resource properties above standard via the proprietary namespaces declarations.

Despite of the rich semantics of the RDF schema, it has a numerous benefits that make implementation and maintenance of the component parsing subsystem easier. There are many convenient open source RDF parsers available for almost any programming language. For example the MOSAIC project uses the rich and extensive library RDF API for PHP. RAP is a software package for parsing, querying, manipulating, serializing and serving RDF models. In spite of the fact, that RDF is usually represented in XML, it does not have a tree-like XML document structure – the document is represented as a set of triples:

- Object
- Predicate
- Subject

This approach makes a machine based processing of RDF very easy and efficient, especially when performing set operations on two RDF documents, which simply means to

perform a set operation on two RDF triples sets. The MOSAIC project uses this property of RDF when building a node manifest that represents all resources that a node provides and/or relies on. More about this is in implementation chapter of this document.

Chapter 4. **System Architecture**

In this chapter we shall discuss what system components The MOSAIC project comprises of, what facilities they provide and how they interact in the global site scope. Later, we shall describe the system node architecture, its parts and their purposes. I will also explain how MOSAIC performs distributed document processing using the Active XML concept behind of it. In the end of this chapter, I will cover the MOSAIC middleware architecture including used protocols, ideas beyond inter component communication and distributed objects invocations.

4.1. Anatomy of the site

4.1.1. Topology

As we are talking about distributed systems, which consist of multiple computers connected over the network, we may represent a distributed system as a graph, where nodes are the computers of a distributed system and edges are the network connections between the computers.

Deciding which topology to use was from the most part influenced by the initial supposition about the primary usage of MOSAIC project. The site should behave as an application cluster, running on the local network, spreading the load between several computers by spreading application components amongst them. This target requires minimal communication latency between nodes. All these assumptions greatly influenced the decision to use a *complete graph* topology of the site illustrated on the Figure 2.

For the sake of simplicity Figure 2 illustrates only 3 nodes, however the number of nodes depends on the site configuration and can be higher. Simply, that means that every site component is directly connected to all other site components. As a new node is dynamically being added to the site, it first establishes connection with the “well known” naming and control agent facility and receiving from it all required information about how to reach all other system nodes.

In discussing the generic MOSAIC site’s topology, yet not limiting ourselves to a distributed system topology term, it is necessary to explain how the MOSAIC site communicates with outer world, i.e. as a web framework, and how it communicates with a web client. MOSAIC is not designed to accept HTTP request directly from the web client’s

software, due to many reasons[14]. Instead, it uses a high performance web server that is able to pass certain types of requests to the MOSAIC sites, managing communication with client by itself. More detailed information on this topic will be given in the chapter dedicated to the implementation of document processing.

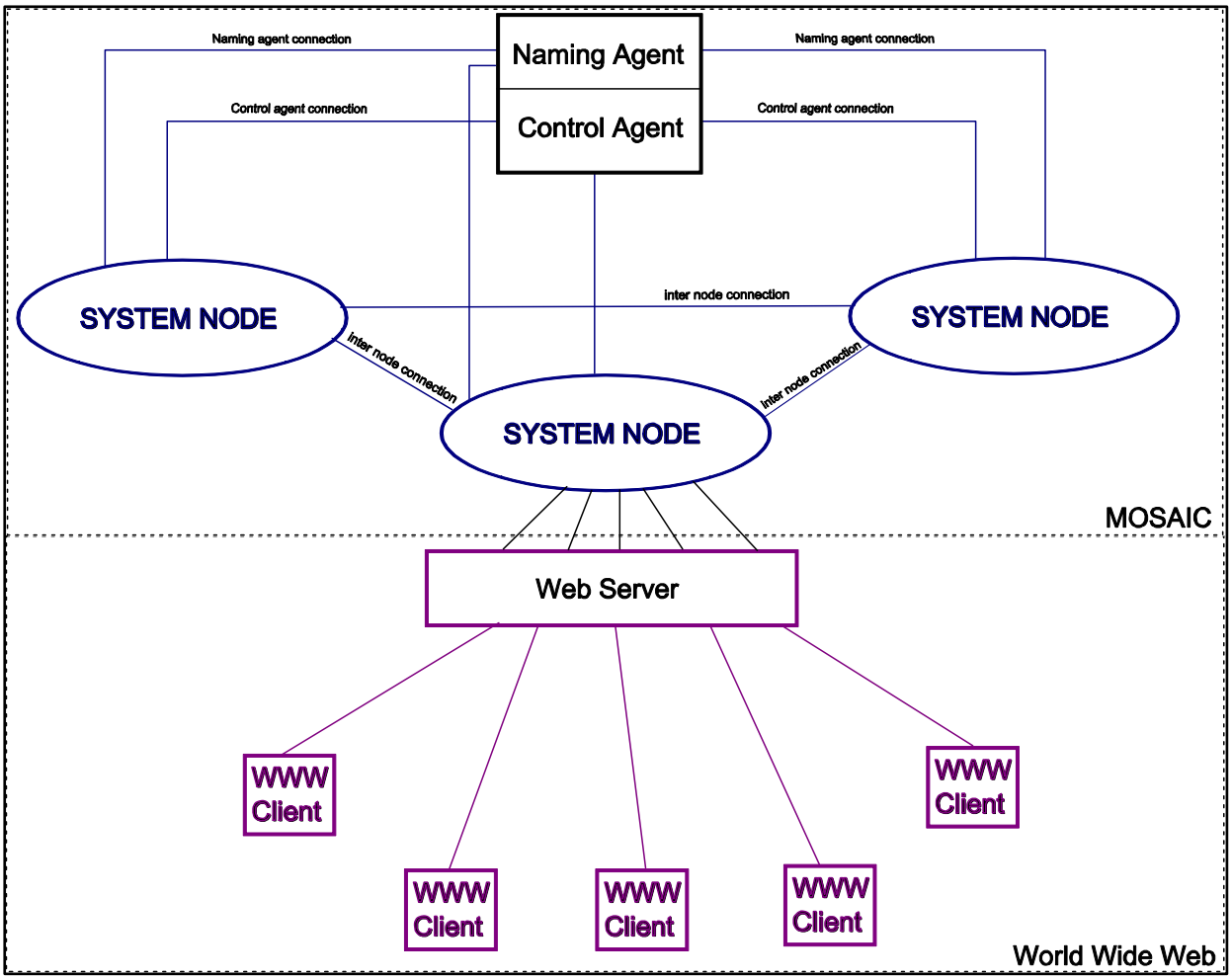

Figure 2. MOSAIC system topology

4.1.2. Naming and control agent

The naming and control agent plays a crucial role in keeping the MOSAIC site together in many ways. It is a separate application that can run on any network connected machine. Each node instance must be advised before its execution on how it must connect to the naming and control agent. Evidently, the agent performs two different, though related to each other functions: naming and site control.

Each node in a site has a unique name that should be given in the node configuration along with the naming server connection information. When the node starts, it establishes a connection to the control agent and sends the "*node up*" message. Other than the node name and

[14] Such reasons include but not are limited to static Web content serving, communication with client that has a slow network channel, full scale HTTP/1.1 protocol implementation. Sometimes approach taken by MOSAIC is referred as frontend-backend architecture. Discussion of these problems is beyond of goals targeted by this document.

DSN connection string with IP address and port information of node server socket, this message contains the node manifest – a RDF document with complete resources and components information. The control agent adds this information into the naming agent data structures and then informs all other nodes about the resources of a newly connected node by broadcasting node RDF manifest to every one of them. The connection is left open for future communication with control agent.

Through the connection to the control agent node may also send a "*node down*" message containing the node name. This message clears the related naming agent caches and also the control agent broadcasts this message to all other nodes in order to inform them about this event. In the case of a node malfunction and unexpected termination, the control agent also detects connection loss and behaves equivalently to the "node down" message reception.

In order to simplify the ease of future code manipulations, extension and heterogeneity, the distributed object naming facility is kept separate from the operations above. A node providing a particular distributed object class, registers with the naming agent a class name, PHP stub code and protocol. Once in place, any client may easily locate which node provides a remote object class, get the object stub code and begin invocations. Despite of naming separation, reception of the "node down" message clears all data related to the distributed objects provided by the referred-to node.

4.1.3. Nodes

Nodes are system parts that encapsulate user application components and distributed objects instances. A node is responsible for establishing and keeping connections to all other nodes, instantiation and execution of applications components', providing inter-component communication by routing and delivering event notifications and finally the handling the remote objects invocations.

4.2. Anatomy of a node

This part of the document is dedicated to the MOSAIC system node architecture and organization.

4.2.1. Introduction

The PHP language interpreter capabilities and properties greatly influenced the design and implementation techniques used in MOSAIC. When talking about *network multitasking*, we can distinguish three basic types of network machine organizations: parallel processes,

threading and multiplexing[15]. Due to the many drawbacks of the parallel processes approach, complexity and what's more important absence of multiple threading executions in PHP, MOSAIC system uses the *multiplexing* mechanism. In the first row this means that the MOSAIC node does not create any new processes as the new requests are coming in.

In the current implementation, a node comprises of three interconnected processes, each performing a specific role: *the request broker*, *the worker* and *the object broker*.

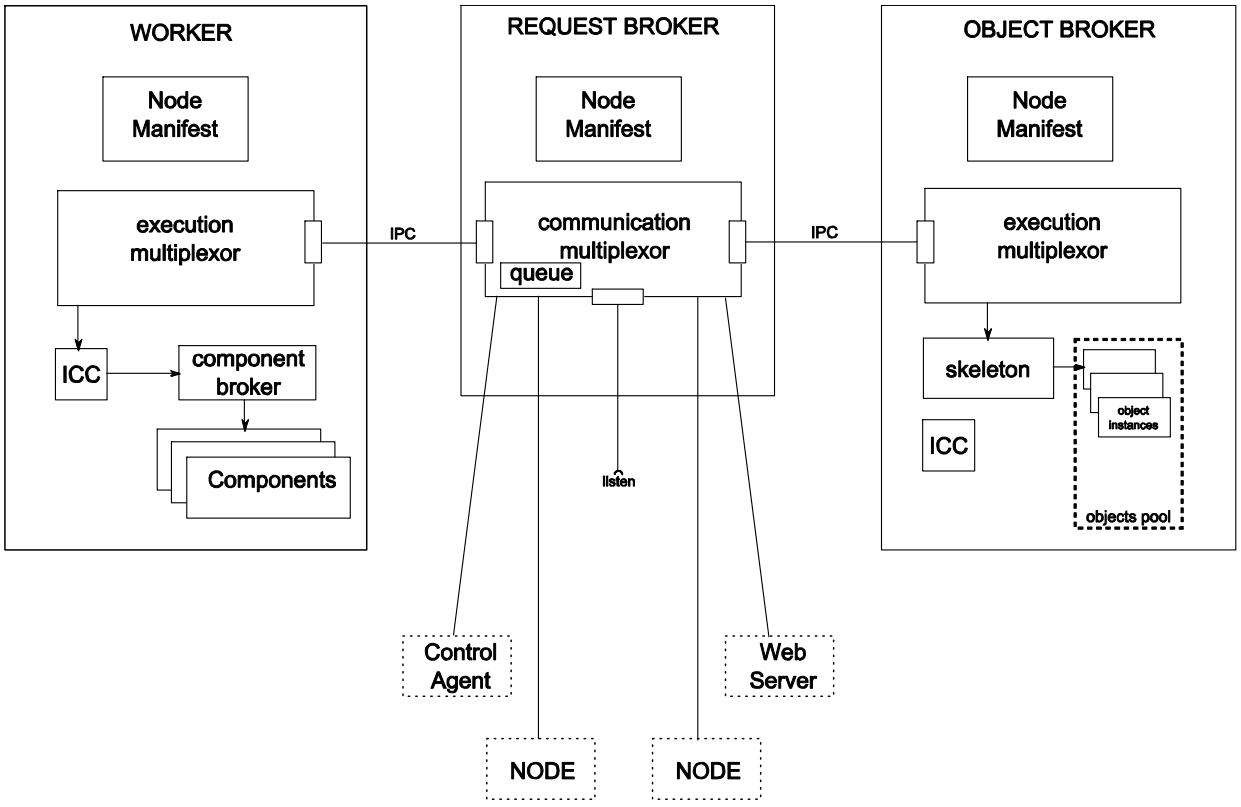

Figure 3. Node anatomy

4.2.2. Request Broker

A request broker is the main node communication agent that keeps connections with external agents, i.e. nodes, web clients and naming and control agent. It also plays role of a request/reply messages multiplexor, queuing and delivering them between site components. In other words, the request broker is the facility that provides non-blocking message based communication with queuing.

There are several types of incoming requests that are served by the request broker, each one of them is directed to the responsible node subsystem: *event notifications* from components,

[15] In different sources, multiplexing can be met under "Event machines" or "State machines" name. In this document I will use just the multiplexing term.

requests for content from web server, *up/down manifests* from other nodes are delivered to the worker process; object invocation commands, i.e. instantiation, method call and garbage collector messages are delivered to the node's object broker.

The request broker is also responsible for the delivery of messages from its own node components to other site nodes and agents. By the type and content of a message, request broker, and with the help of the node manifests information and naming agent, it finds out which agent must receive it and sends the message through the previously established connection. If a connection is not yet established, the request broker establishes and keeps such connection. This is done using the target node's connection information from naming agent and a special handshake procedure.

4.2.3. Object Broker

Object broker it the node process that accommodates distributed objects instances, skeleton code and processes object invocations. PHP language capabilities also made possible creation of a generic universal object skeleton, which is a part of object broker code. Object broker gets following requests: *object instantiation request* and *method invocation request*.

Object instantiation request contains name of the remote object class. Object broker creates and instance and returns object reference (URI) to the callee.

Object method invocation request contains object's URI received from instantiation request, name of the method and marshaled parameters. Binding and unbinding garbage collector messages is encapsulated into the __bind() and __unbind() method invocations.

4.2.4. Worker

A Worker is the process that accommodates and runs business logic application components. A Worker process comprises of three main parts: request execution multiplexor, inter component communication facility and the component broker.

Request execution multiplexor is the most advanced subsystem that allows parallel processing of web document requests made to the MOSAIC site in the global scope. When component fires a synchronous boomerang event notification it is blocked until the boomerang event returns with data from all recipients. In this situation the request multiplexor starts the execution of another request pending in the request broker queue. When the event notification returns, the multiplexor continues execution of the original component code. This approach reduces the amount of time a worker wasting by waiting for a response from other nodes to the minimum.

In order to synchronize the execution of requests, MOSAIC defines the *execution context*, shared across all nodes, and is implemented with the help of distributed objects facility. The request execution multiplexor is responsible for performing a *context switch* and *context release* operation to synchronize request execution with the worker state. A more detailed explanation on the multiplexor algorithm and implementation techniques will be given in the implementation chapter.

The component broker is responsible for the management of component instances. Each component is represented as a PHP class and may be defined as a *stateful* or *stateless* component. Stateful component instances are singleton in the context scope, stateless component instances are singleton in the worker process scope. The component broker's job is to deliver a reference of a correct component instance to the Inter Component Communication facility based on the current worker context. This delivery operation may include loading of a relevant component class code and the preparation of some resources, of which a component depends on and after instantiation of the component class instance. After this, a component class instance is placed in the component broker pool for future usage. Components are removed from this pool during the context release operation.

4.2.5. Resources

Before talking about the components, we need to shortly introduce to the MOSAIC system resources. Resources are system entities that are required or provided by components. Framework distinguishes *data resources*, which are: *include resource*, *class resource* and a *kernel module resource.* The other class of resources is processing resources that are used as a medium to encapsulate ICC messages, such as event notification invocations. These resources are the *event resource* and the *tile resource.*

Include resource

Include resource is a generic PHP file, that is usually needs to be loaded before the component instantiation. This file may contain some constant definitions, package and/or component configuration initializations etc. It should not contain instantiatable class definition as for class definitions more specialized resource type is available.

Class resource

A class resource is a resource that represents a PHP class. There are several reasons why classes are treated as a special resource type.

First is that classes are loaded on demand, so the system needs to have a mapping of a class to the file.

Second is that the remote object class resource contains a declaration of two files, one of them is the stub and the other is the actual object code representation. Another reason is that MOSAIC provides limited on demand code transfer functionality between nodes and the class resource instance behaves as a container where the class code is encapsulated.

Kernel module

The kernel module is a virtual resource, which represent a component's capability of being registered as a kernel module. A kernel module is a technique that allows a component instance to be available by references in the global scope in order to perform direct method calls instead of using the ICC communication primitives.

Event

As mentioned before, triggering the event notifications is the only valid way how components communicate with each other. Event execution is a generic method to run a procedural component code from any place in the code. Events are named and delivered to all components on all nodes that manifest themselves to receive an event of a given name. An event can be executed synchronously or asynchronously. A caller should specify whether it fires the event in synchronous or asynchronous mode.

Tile

Tile resource is used as an active element of the Active XML representation. While the event is the more generic way to run a component's code, tile processing is a special case of code execution, invoked by the parser subsystem during the Active XML tag processing. Tile processing is done asynchronously to provide more parallelism in Active XML document processing. From the implementation point of view, tile resource processing acts like an asynchronous boomerang event notification.

4.2.6. Components

A component is a PHP class that represents and encapsulates business logic of an application. For better representational and organizational purposes components are being organized into *Packages*. Each package should be placed in a separate folder containing the package files and RDF package manifest file, of which defines generic information about the package using the DCMI [8] tags, such as creation and modification date, package name, copyright and author information. The package manifest file also contains component

definitions which include unique component system identification in URN format [9], a class name that contains the component code and provides the required resources of the component.

Required (dependent) resources section contains resources that should exist in the site in order to instantiate and execute the component. It may be the existence of an event notification handler (event dependency), tile resource processing handler (tile dependency) or to be as simple as including a generic PHP file (include dependency). In other words, by putting dependencies in this section the component informs the system which resources it is going to use a lets the system deal with it before instantiation of the component.

Provided resources section is used to tell the system what resources a particular component is providing. The component may provide all types of resources with an exception of include resources provided that the event and tile resource is equivalent to the component's subscription for the processing of these resource types. It is necessary for the component to specify a component class resource.

In the package manifest each component is also specified to be stateful or stateless. This setting influences the component broker component instantiation behavior: stateless component is instantiated only once, thus all requests are processed by the same instance; stateful components are instantiated once per each execution context, thus all requests in the same context are processed by the same component instance.

Diagram on

Figure 4 illustrates RDF schema for component interface manifest document.

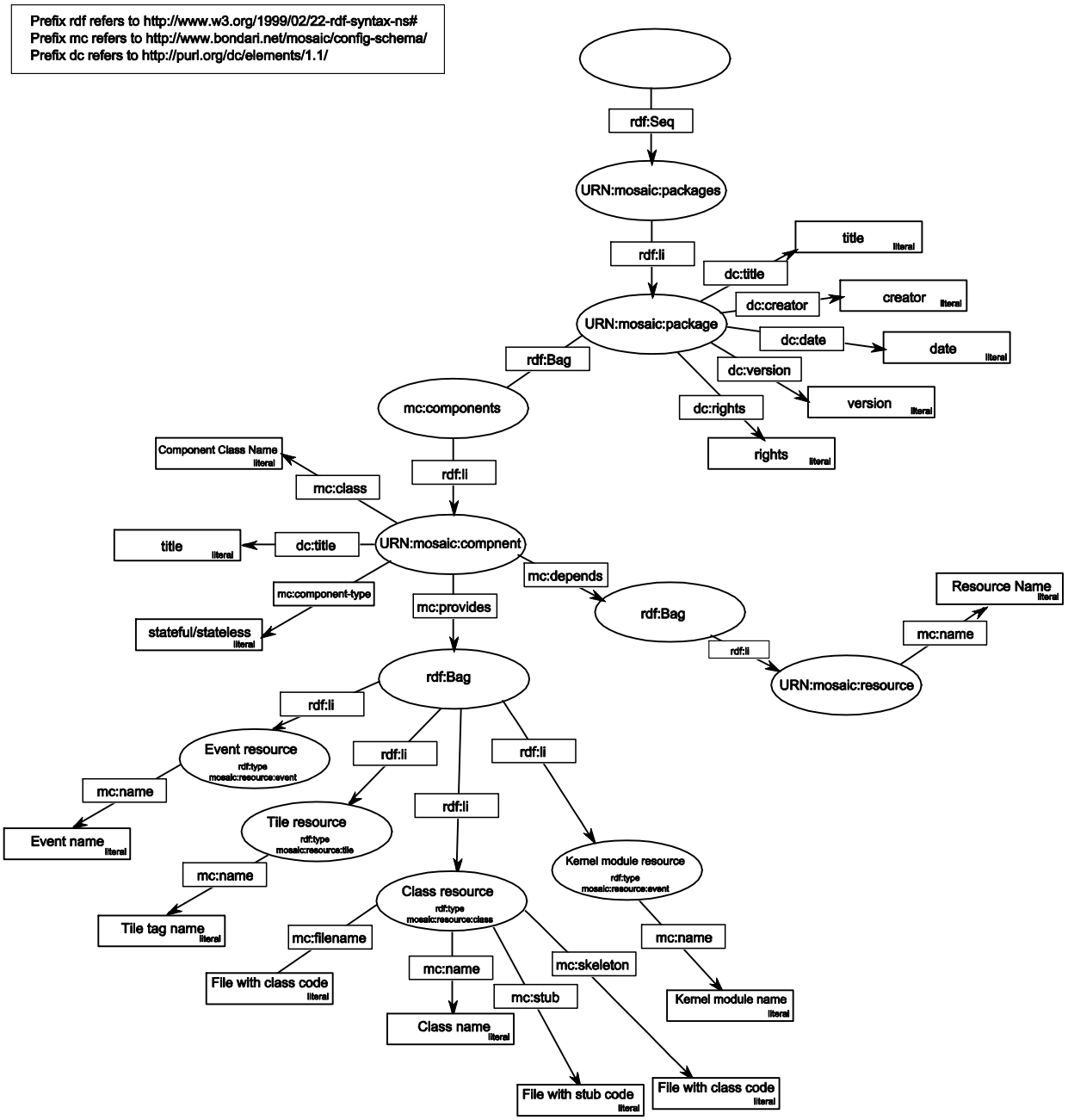

Figure 4. Visualization of package manifest (RDF).

4.3. Active XML

Active XML is a concept that is used for distributing the document processing in MOSAIC framework. In this part I will define and explain this concept.

4.3.1. Introduction

Although the term Active XML is used in several projects, in MOSAIC this name came up naturally during the initial planning of MOSAIC system specifications. Main idea behind the distributed XML document processing model is that an XML document is being split into some previously defined and named parts and these parts are sent in parallel to different nodes for modification. In the Web document system this modification process will usually (though not necessarily so) result in building up a bigger XML fragment, extending or replacing the original fragment with more data. This developed fragment is returned to the original document and is being inserted instead of the fragment that invoked the process. Such a process may be done recursively so that after the buildup step, the document undergoes same process again. When we take in consideration the XML tree-like structure [10] of the document, it makes the most sense to use a tree branch as the fragment mentioned above (i.e. use tags to surround fragments of the document intended for distributed processing).

In MOSAIC these tags are called *tiles*. To identify these tiles during processing, MOSAIC defines an XML namespace http://www.bondari.net/mosaic/system-resource/tile

4.3.2. Distributed processing of AXML document

Distributed processing of the document is performed by 2 subsystems: *parser* subsystem that parses XML documents and passes *tile resource requests* populated with these fragments to the ICC subsystem, of which is responsible for locating components that process these tiles and the delivery of these requests to the correspondent components. The component always receives a tile name and the complete DOM document the tile node (i.e. all XML data contained inside the tile tag.).

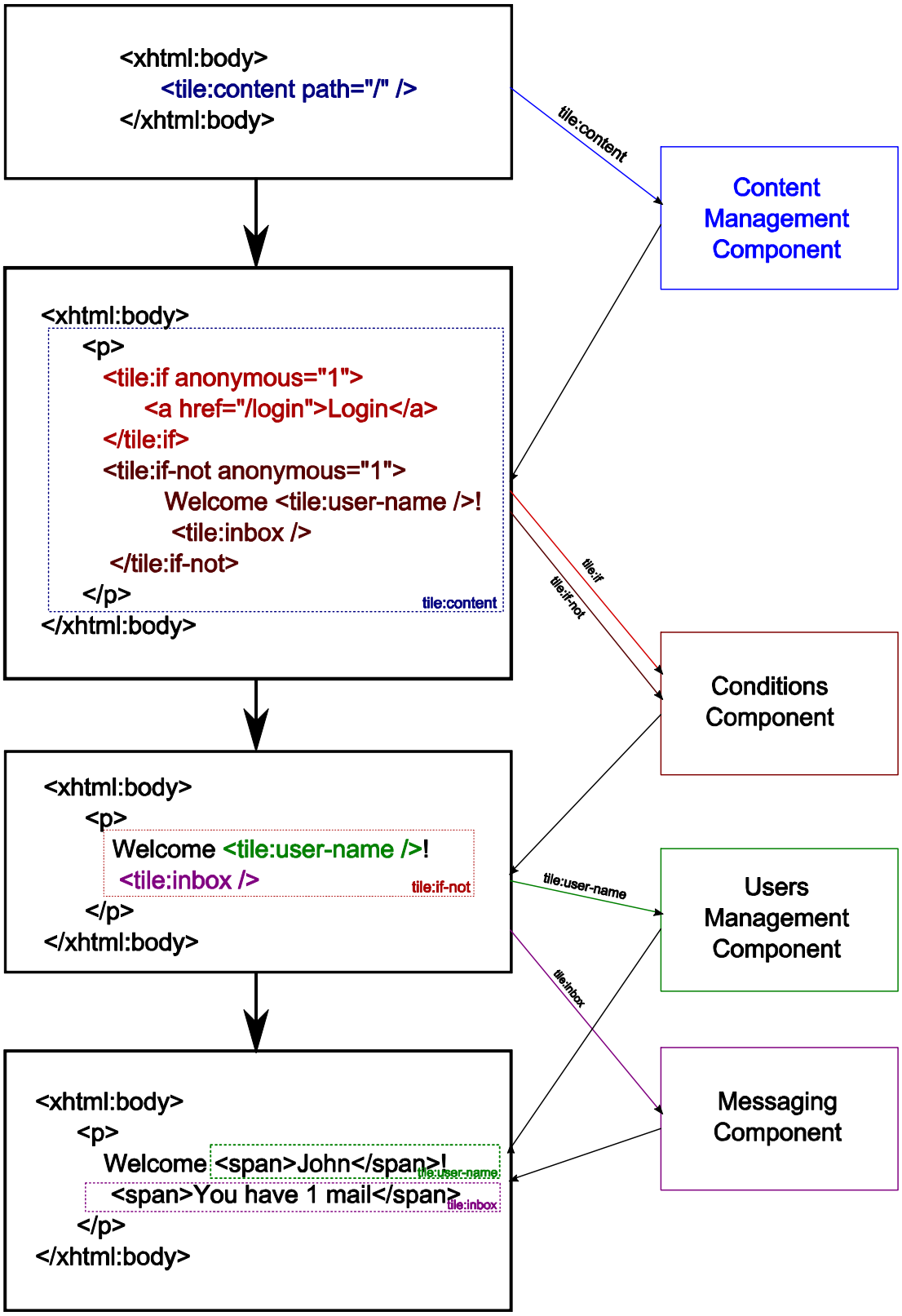

Figure 5. Active XML processing example

In order to process all tiles in parallel, ICC uses the same mechanism for the delivery of these requests as asynchronous boomerang event notifications. Components subscribe to these types of requests by providing a tile resource with the tile name in the component's "provides" section of the package manifest.

When the request returns, the parser replaces the original document fragment with the data returned from the processing and recursively parses this fragment again until there are no tiles left for processing.

Parsing is finished when there are no tiles left in the document. The result of the parsing is an XML document that does not contain tiles, which is delivered to the web server.

Chapter 5. **Implementation**

5.1. Middleware

5.1.1. Communication protocol

As mentioned before, MOSAIC server is not designed for communication with client directly, that is why it does not support and implement HTTP protocol. Instead, it was decided to approach this problem from the same vector as FastCGI does. One of the FastCGI benefits above the standard CGI approach is that it is designed for distributed computing: Companies can run their FastCGI application on a different machine from the one on which they run their Web server. Distributed computing is a proven technique for scaling, linking to existing corporate systems, improving system availability, and improving security via compartmentalization, such as firewalls [11].

Unfortunately, the FastCGI protocol is quite complex for implementation in PHP and such work is beyond the scope of this project, although the FastCGI multiplexing support would naturally be supported by MOSAIC. It was decided to utilize the Simple Common Gateway Interface protocol [12]. SCGI protocol was developed Neil Schemenaue, to replace a FastCGI protocol where ease of implementation is needed. As the FastCGI protocol, SCGI is supported by all major web servers.

In order to effectively employ SCGI protocol with the MOSAIC distributed environment it was necessary to extend the SCGI protocol for MOSAIC needs. I have named this protocol Multiplexed Common Gateway Interface (MCGI). MCGI built to be backwards compatible with SCGI, so MOSAIC natively accepts SCGI requests from the HTTP server, but uses the MCGI protocol version internally.

The MCGI packet format is defined same as SCGI packet format, but unlike SCGI protocol, the MCGI protocol defines the response format, which is structured the same as a request. A request and response consist of a number of headers and a body. The format of the headers may be represented as:

```
headers   ::= header+
header    ::= name NUL value NUL
name      ::= notnull+
value     ::= notnull*
notnull   ::= <01> | <02> | <03> | ... | <ff>
NUL       ::= <00>
```

Duplicate names are not allowed in the headers.

The first header must have the name "CONTENT_LENGTH" and a value that is a nonempty sequence of ASCII digits giving the of the body length in decimal.

There must also always be a header with the name "SCGI" and a value of "1" for SCGI protocol and "2" for the MCGI protocol. If a request is in SCGI version of protocol, the response format is not strictly defined and the socket must be closed after the response was transmitted. In the case of the MCGI protocol, the socket may be kept open for future communication.

In order to maintain packet multiplexing synchronization, each request and response must also contain the "REQUEST_TOKEN" header, a unique alphanumeric identifier of request (in case of request packet) or (in case of response packet) identifier of originating request. In order to effectively distinguish responses from requests, the response MCGI packet must have a "RESPONSE" header with the number "1" as for value.

In order to facilitate the transition from CGI, standard CGI environment variables should be provided as SCGI headers. The headers are encoded as a *netstring*. Netstring encoding is explained in SCGI protocol specifications [12]. The body is sent following the headers and its length is specified by the described above "CONTENT_LENGTH" header.

5.1.2. Requests

Before going deeper into the implementation details of different MOSAIC subsystems, it is necessary to quickly introduce and enumerate system requests. A request is a middleware entity that represents direct correspondence with the MCGI protocol packet (i.e. directly convertible from and to MCGI protocol packet).

Every request is assigned a *unique identifier* across all nodes. This unique identifier request is used to clearly identify a response to the request during the multiplexed requests execution. The unique identifier is constructed from the node name, a micro timestamp and random seed. The selection of this criteria guarantees uniqueness of the request identifier across all nodes.

With an exception of HTTP_CONTENT requests, all requests are tagged with the *context identification string* that allows nodes to synchronize requests execution context.

There are several types of requests, each playing different role in different parts of the system.

Request for content generation

This is a request that encapsulates SCGI server requests and is designated as `HTTP_CONTENT`. Request contains CGI environment data, HTTP request headers and POST data. The response contains the XHTML document and HTTP output headers. Requests of this type trigger creation of the new context.

Naming and control agent requests

`NODE_UP_MANIFEST` requests are sent by the node when connecting to the site. This request is basically broadcasted by control an agent across the whole site.

Parameters: node name, node connection information and node RDF manifest document.

Response: exception if node name is already occupied

`NODE_DOWN_MANIFEST` request is sent either by the node during a graceful shutdown or by a control agent when it detects unexpected a node failure or disconnection. This request is also broadcasted across the site.

Parameters: node name

Response: undefined

`NS_NODE_RESOLVE` request is delivered to the naming facility in order to get information how to connect to the specific node.

Parameters: node name

Response: node connection DSN, exception if node is unknown

`NS_REMOTE_CLASS_MANIFEST` request is delivered to the naming facility from node object broker.

Parameters: node name, class name, stub code

Response: Exception if class is already registered

`NS_REMOTE_CLASS_RESOLVE` request is resolved by a naming facility. It returns a node name that hosts remote classes of a given class name.

Parameters: class name

Response: node name or exception if class is unknown

Distributed objects

`REMOTE_OBJECT_STUB` request for stub code; may be resolved by the node that hosts the remote object class or by the naming facility.

Parameters: class name

Response: stub code or an exception

`REMOTE_OBJECT_INSTANTIATE` request contains an instantiation request and is delivered to the corresponding object broker.

Parameters: class name, marshaled constructor parameters

Response: URI of the object or an exception.

`REMOTE_OBJECT_CALL` request contains a remote object invocation data: object URI, method name and marshaled parameters. Response contains the return value of the invocation or an exception.

Parameters: URI, method, marshaled parameters

Response: return value or an exception

Inter component communication

`RESOURCE` request is used as a container for processing resources (tiles and events) delivery.

Parameters: serialized resource data

Response: serialized processed resource data or an exception

Nodes synchronization

`RELEASE_CONTEXT` is a low priority broadcast message that is initiated by the node that created a new distributed context. This message tells other nodes that they may release resources associated with given context.

Parameters: Context ID

Response: undefined

`PING` request is used during nodes handshake.

Parameters: node name

Response: node name

5.1.3. Remote Exceptions

MOSAIC middleware libraries implement remote exceptions. This is a completely transparent mechanism provided by the middleware that passes application level exceptions raised by the callee to the caller, suppressing the exception effect in callee process and raising exception on caller. Remote exceptions are captured during remote object invocations, ICC calls and some of the system calls, such as naming and control agent registration and lookups.

All application exceptions that are meant to be transferred between application components must be derived from the `RemoteExceptionThrowable` class. This class is able to encapsulate a `RemoteExceptionContainer` object instance which represents a snapshot of the exception captured on the callee side. After the capture, the exception container is being serialized and into the `Response` object, which is delivered back to caller. When a caller tries to read data from the response, the middleware raises the exception of the same class populated with the remote exception data.

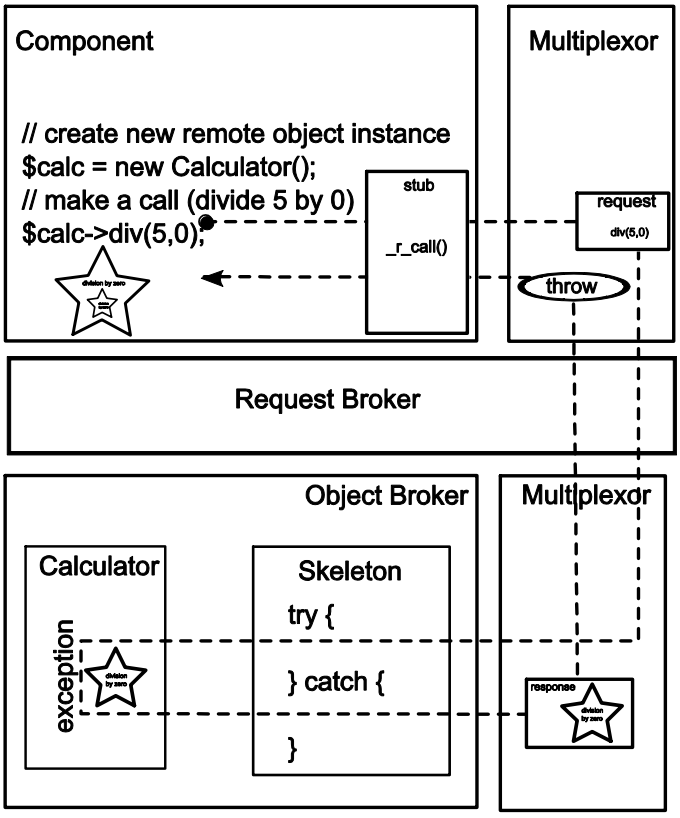

Figure 6. Remote exception example

Figure 6 illustrates an example of how remote exceptions function during the synchronous remote object invocation. The application component instantiates the remote object stub and makes a call to divide 5 by 0. The stub packs the invocation into the request packet and passes it to the multiplexor. The multiplexor delivers the request to the responsible

node's object broker multiplexor. The multiplexor passes the request to the object broker and the object broker skeleton code runs the object code in `try ... catch` block, attempting to catch only the `RemoteExceptionThrowable` exception class. If an exception is caught, the object broker encapsulates the exception into the exception container and places it into a response. The response is delivered to the callee multiplexor and just before returning the call result; the stub exception is being thrown. The exception has the same exception message and the remaining data is available from the exception snapshot.

5.1.4. Distributed Objects

The distributed objects middleware functionality contains of several relatively independent subsystems and libraries: stub base class, object broker (including unified skeleton code and garbage collector) and naming facility.

Stub base class named `RemoteObjectStub` implements all possible functionalities that might be needed in the stubs implementation: explicit and implicit binding mechanism, remote constructor, remote destructor, remote method invocation and finally the remote properties setter and getter.

Object broker is the node subsystem that accommodates objects' instances, plays a role of skeleton i.e. performs parameters unmarshalling, method call and response marshalling and delivery. The object broker also implements a part of the server-side Garbage collection

Naming facility keeps remote class name-to-node reference and stub code. The naming facility is used by request brokers while routing the invocation requests to the responsible node.

5.2. Node

5.2.1. Inter component communication

As explained before, inter component communication facility supports 3 types of inter communication modes: synchronous event, asynchronous event and tile processing. Each one of these functionalities is invoked by calling a corresponding ICC method.

Method `ICC::fireEventSync()` is called for synchronous event notification delivery. Using the `CompoentBroker::getProvidingComponentInstances()` call it gets all component instances[16] that are subscribed to the event and starts delivery of the event, first to local components and after to remote components. Delivery of the notification to the

[16] The component broker actually does not return an instance of a remote component. Instead, it returns a remote component stub object that contains component's info data and the node name that hosts the component. ICC code detects such stubs by testing an instance to support ifRemoteComponent interface.

local component is straightforward – `Component::onEvent()` method is called. Events delivery mechanism and strategy are illustrated on Figure 7. Delivery to remote component is done by encapsulating the event into the generic request-for-resource container and executing multiplexor request processing synchronously. Synchronous event execution may by also aborted by the callee, by executing the `Event::cancel()` method. Execution stops and ICC returns control to the caller.

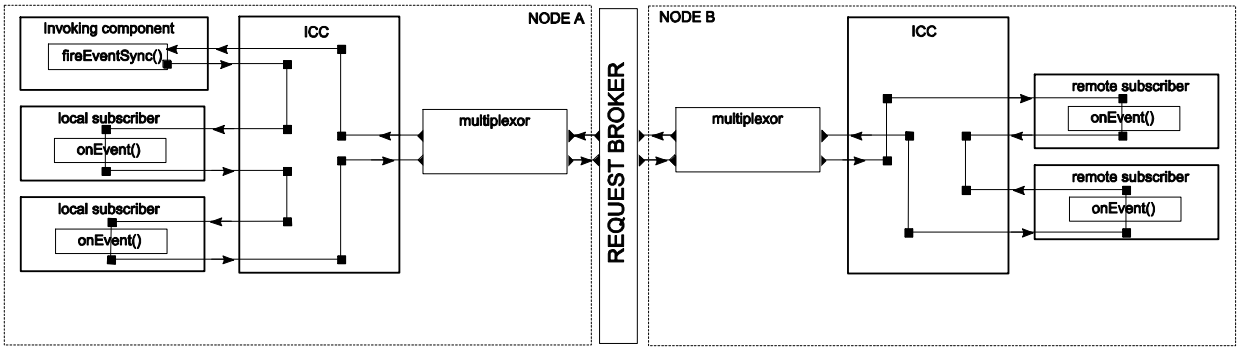

Figure 7. Synchronous events execution

The method `ICC::fireEventAsync()` is called for asynchronous event delivery. In general, asynchronous event delivery is quite different from synchronous. To extend the level of execution parallelism, asynchronous events are delivered to all remote subscribers independently and simultaneously. In a situation when a caller does not require the event with data to be processed after it returns, this fire-and-forget concept is straightforward and works well. As illustrated on Figure 8, events are sent to all remote recipients in parallel and the returning data is omitted. In order to provide non-blocking `fireEventAsync()` behavior, local event execution is performed only upon asynchronous remote events return.

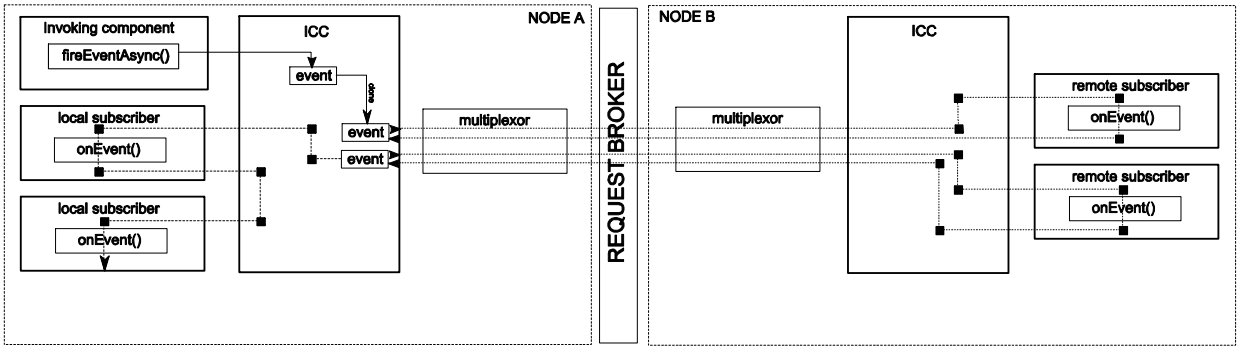

Figure 8. "Fire and forget" asynchronous event execution

However, if data should be processed after, the programmer needs to program a *merging routine* that combines the data returning from the remote components back into the original event object instance. Such an operation is not always clear and appropriate and it is up

to the programmer to choose whether he wants to use synchronous boomerang events or asynchronous events with merge and callback implementation illustrated on Figure 9.

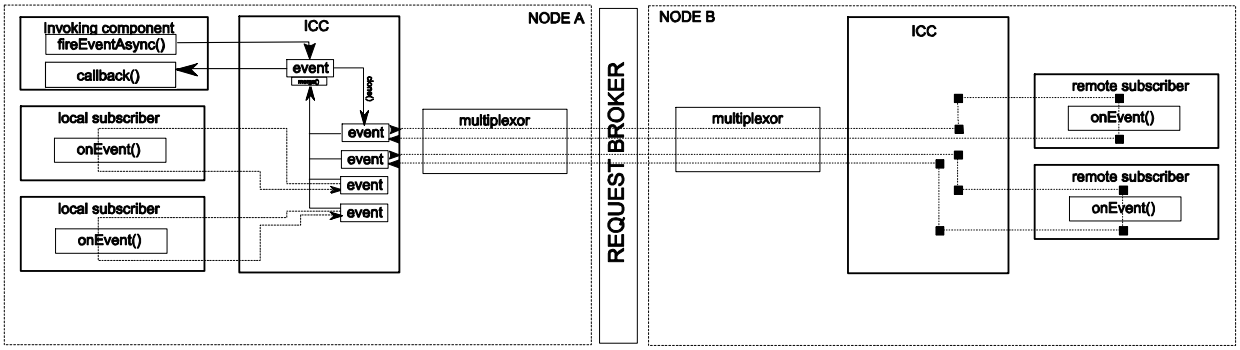

Figure 9. Asynchronous event execution with merging.

With asynchronous events a caller has the ability to process the event data after the execution finishes by setting the callback on the event object instance. To synchronize and control asynchronous event execution, the event object actually contains a *callback stack*, where the caller callback is placed on the bottom of the callback stack. ICC distinguishes several situations when asynchronous delivery is demanded:

There are **no remote components** amongst the event subscribers. In this situation taking in consideration lack of threading support ICC runs the event synchronously.

There are **only remote components** amongst the event subscribers. ICC checks if the caller defined a **callback** to process an event upon return. If so, a callback stack is padded with empty function calls, one for every remote component, except one. Then for each remote recipient an event is encapsulated into the request-for-resource container and handed to the multiplexor for asynchronous execution with a special ICC defined callback function to be called upon event return. This function calls the data merging function and pops one callback. The last event returned will pop the original caller callback.

If no callback is defined, the ICC just sends the event to all recipients via the request multiplexor asynchronous request execution with no callbacks defined (Figure 8). This operation will just deliver the event to all subscribers with no actions performed on the caller node after.

There are **local and remote components** amongst the event subscribers. This situation is very similar to the previous one, except for the callback stack will have one more callback function that runs the event locally after all remote events returned.

5.2.2. Worker multiplexor

The worker process multiplexor facility is one of the core subsystems in the MOSAIC framework. The main role of the multiplexor is to allow a worker to avoid blocking while making synchronous calls to other nodes. Along with the node context switching mechanism, the multiplexor simulates non-preemptive kernels behavior. Using the kernel development terminology we may treat synchronous execution as a syscall, the multiplexor facility as a scheduler and a component as a process.

Multiplexor represents a simple event machine that communicates via a single UNIX socket with request a broker process. It also contains a request and response pool – a queue which holds all active (not resolved) requests and responses. The multiplexor also has 2 main methods that drive the multiplexing process. Method `processOutgoingRequest()` is called by a component to pass any request for execution. The request object necessarily contains a flag whether the request should be run in synchronous or asynchronous mode. The second method is `multiplexor()`, a main multiplexor loop. Multiplexor loop uses `select()` syscall to distinguish "read from request broker" and "send to request broker" operations.

The initial state of the multiplexor is empty request pool and the multiplexor is blocked in `multiplexor()` loop waiting to **read** data from request broker (Figure 10, Diagram A). When a new request is incoming, the multiplexor signals a node to create a new context and passes the request to ICC for execution. During the execution, components may pass a request to the request multiplexor. If a request is passed for *asynchronous* execution, the multiplexor just adds the request to the outgoing requests queue and returns it immediately (non-blocking behavior). Things become more interesting when the component performs a *synchronous* request execution (blocking behavior). This operation adds a request to the outgoing requests pool and the `multiplexor()` loop function is called again (Figure 10, Diagram B). The only parameter of this function is the request identification ID *upon which multiplexor must return when response is available* called `$returnUpon` (Figure 10, Diagram C).

Due to the fact that outgoing queue is not empty, the loop is now waiting to **read** or **write** to the request broker, whatever is allowed to do. If request broker is ready to receive data, the multiplexor sends a first request from queue. If the request broker is ready to send the data, the multiplexor receives a packet that may be either a new request that will be processed or a response to the request sent before.

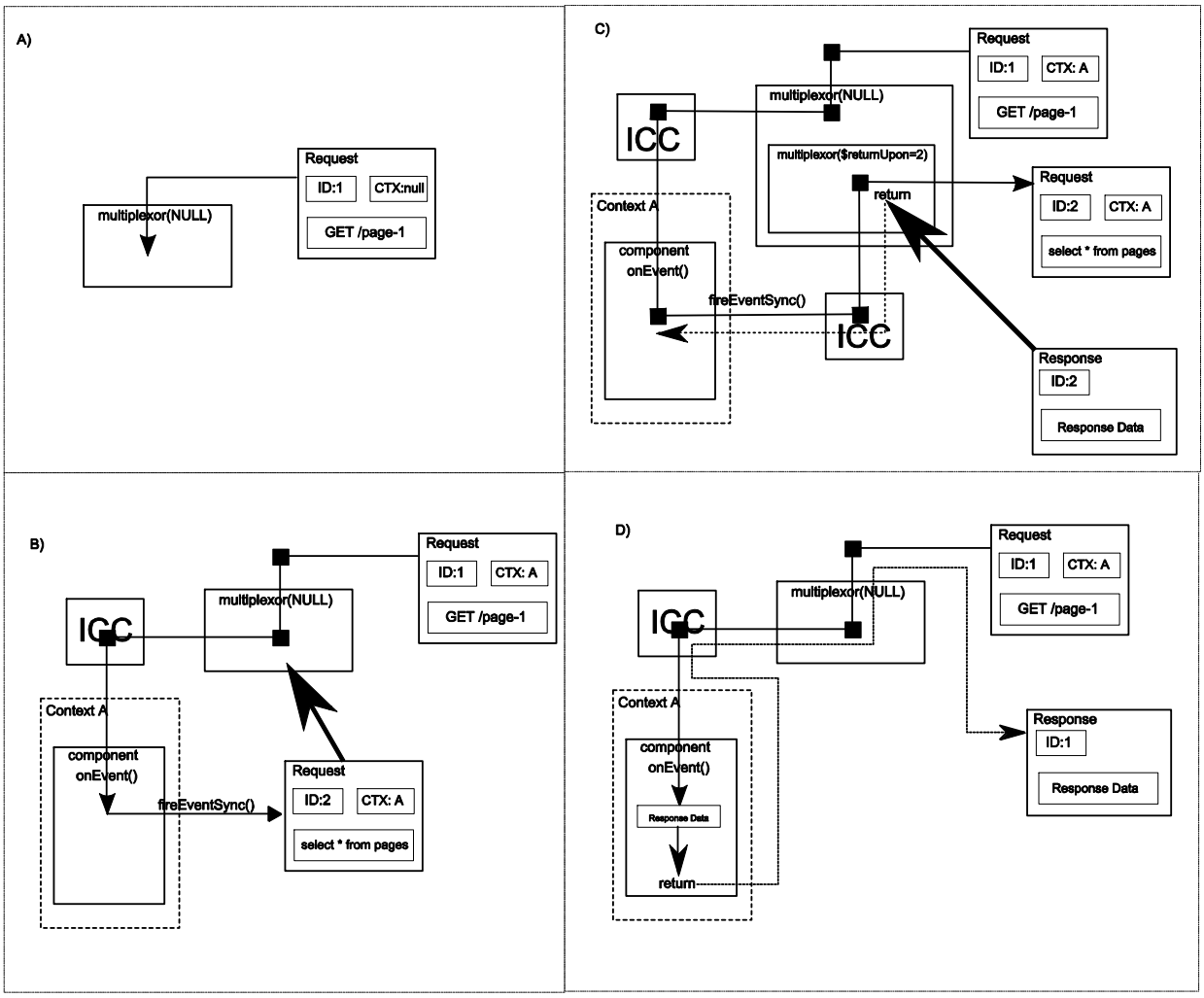

Figure 10. Requests multiplexor. Synchronous request execution.

If the response is the one we are waiting with `$returnUpon` parameter, the multiplexor synchronizes the worker's context, returns the response and execution of component's code continues (Figure 11. Request multiplexor synchronization (matched response))

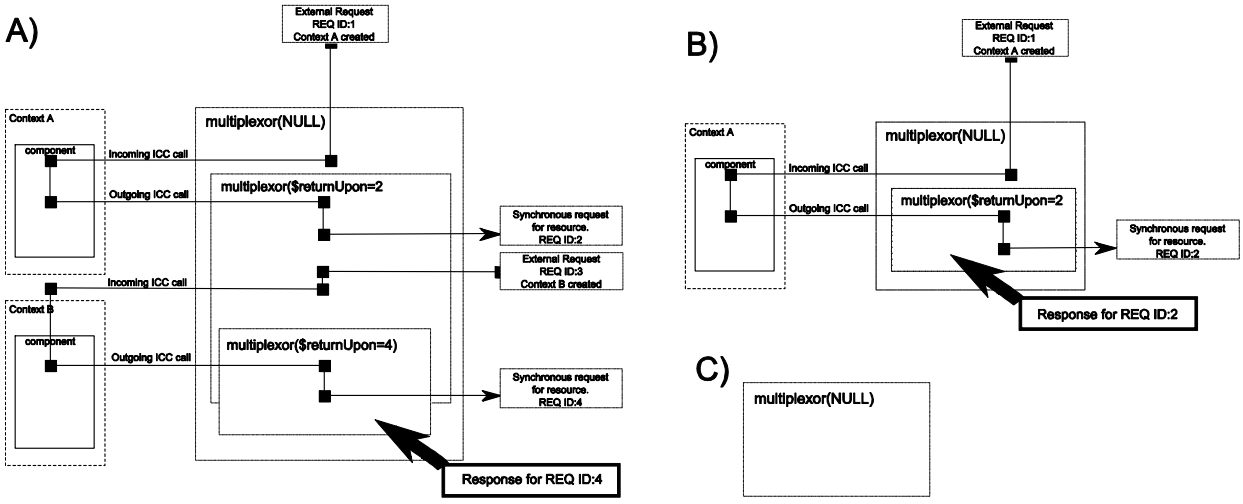

Figure 11. Request multiplexor synchronization (matched response)

If response is not the one we are waiting for, it is placed into the pool of responses for future usage (Figure 12, diagram A and B). After the request broker communication cycle, the multiplexor verifies whether the pool of responses contains the response we are waiting for with `$returnUpon` variable (this response could arrive earlier when we were NOT waiting for it as shown on Figure 12, diagram A) and returns the response immediately (Figure 12, diagram C).

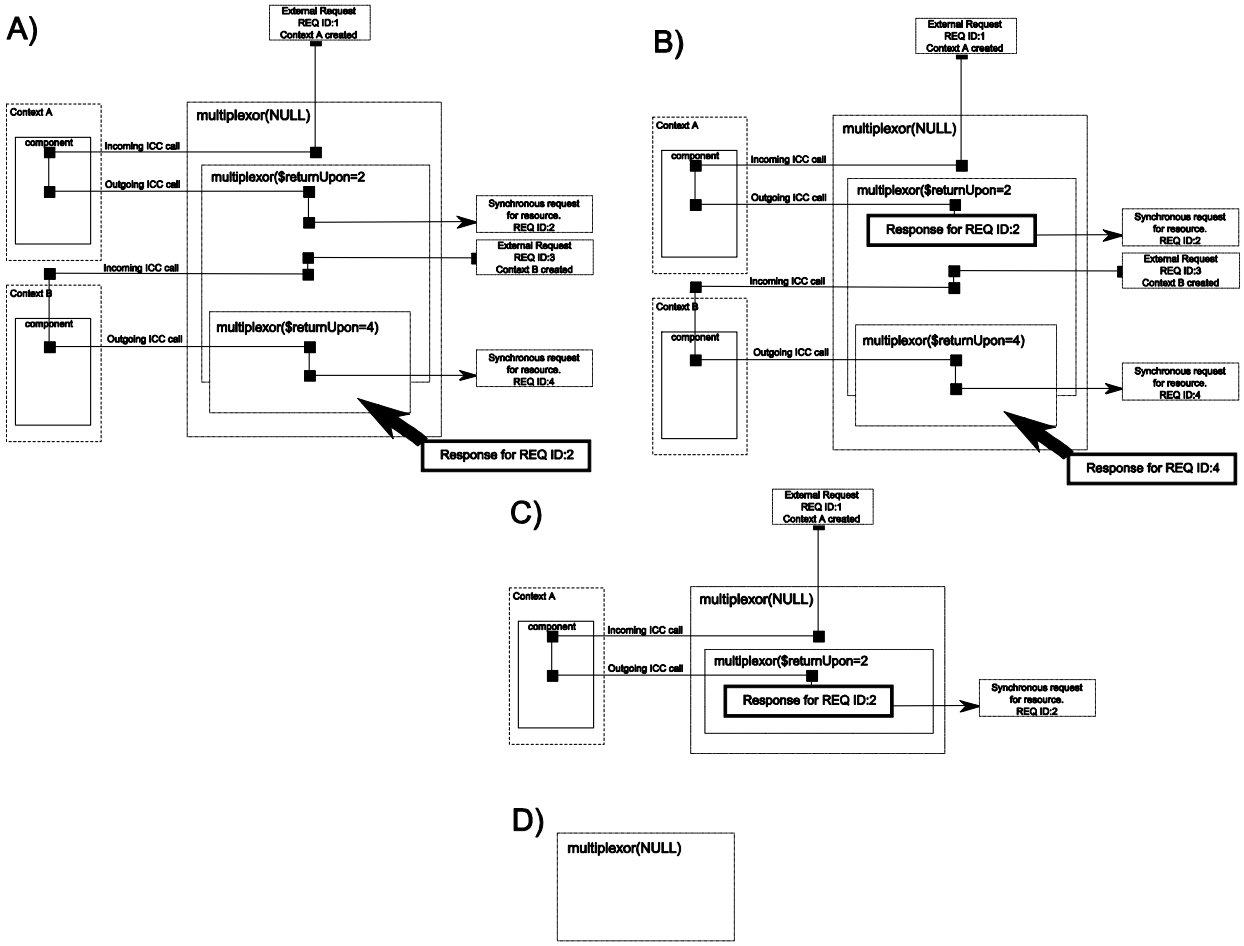

Figure 12. Request multiplexor synchronization (unmatched response)

The advantages and disadvantages of this implementation are as follows. The system creates a multitasking environment with a context switch in the single process, and a single thread execution situation simulating multiple stacks. The biggest disadvantage is the recursively increasing depth of the `multiplexor()` function execution stack size. If the design of the application allows for placing components on multiple nodes so that cycles between nodes do not happen, the depth of the recursion is less or equal to the number of concurrent web requests that may be easily limited to the appropriate number. However based

on the context identification of the request is possible to build a cycle detection mechanisms that would control depth of `multiplexor()` stack by reducing number of concurrent requests. In the worst case the system could gracefully cancel a cycled request execution using the remote exceptions mechanism.

5.2.3. Execution Context

As briefly mentioned in the Worker multiplexor article, execution context helps to synchronize multiplexed pseudoparallel execution of requests.

Each new request received by the MOSAIC site is processed in the specific context that contains data, required or created by components while processing the request. Context data that are used by a single node are called *local context data*. In general, local context data consists of stateful component instances. This data is not available to other nodes except for the node that this data resides on.

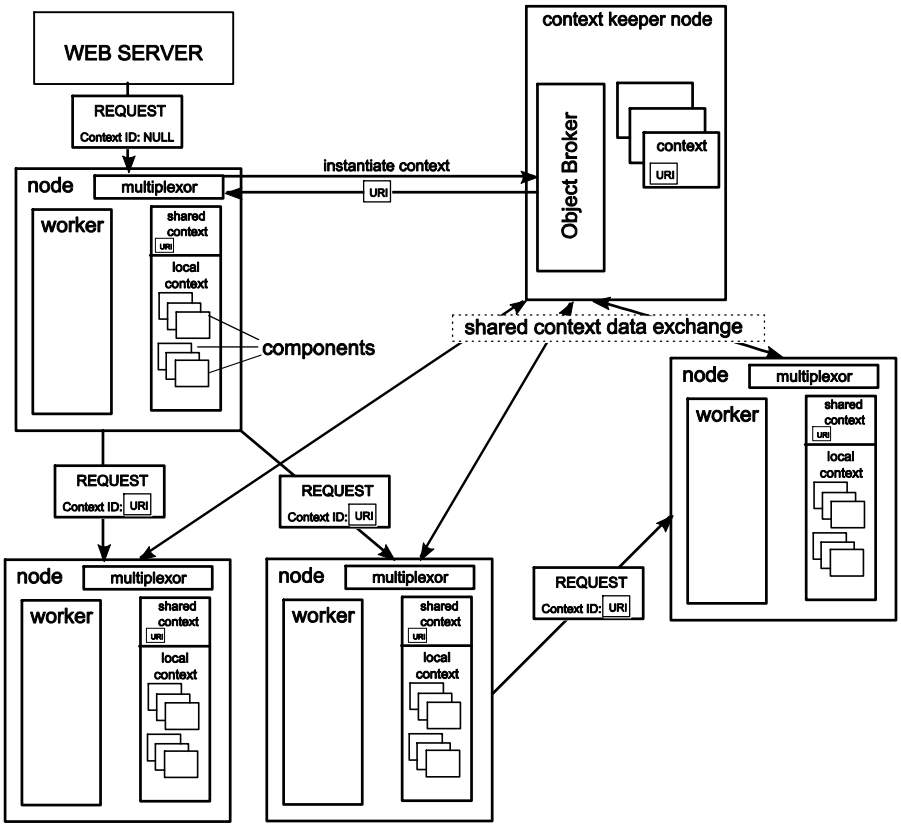

Figure 13. Execution Context

Shared context data is available for all nodes employed in the request execution at the same time. For this purpose MOSAIC uses the distributed objects functionality, so that shared context is simply implemented as a remote object instance and context identification string is

the remote object instance URI. A node that accommodates context distributed object instances is referred as *context keeper* node.

As illustrated on Figure 13 a new request coming from a web server is not tagged with any context tag. That signals the request multiplexor to create a new remote context object and get its URI. The worker creates an empty local context pool and begins execution of the request. Any stateful component instance that will be created will be placed into this pool. During the request execution any outbound request is always tagged with an active context tag so that the node it arrives on may access the same shared data and also synchronize execution of requests with the caller (i.e. process all request coming from a particular caller in the **same** context with the same component instances, thus preserving state of components inside the context).

Context release operation is performed after the response that triggered the context creation is sent to the request broker for delivery to the web server. A context release message is broadcasted to all nodes and has lowest priority. This message also does not require any response and used to inform all nodes that they are allowed to free and destroy all stateful data and resources associated with broadcasted context ID. The context object is destroyed automatically by the distributed objects garbage collector after release of the context on all nodes.

5.2.4. Request broker

The request broker is a classic network event machine implementation that exploits the fundamental operating systems capability to wait for data input and output on multiple data resources. In particular these resources may be network sockets or IPC entities. To unify the network communication event loop and IPC event loop, the request broker uses the UNIX socket for IPC inside the node with the worker and the object broker. Using the `stream_select()` system function the request broker is constantly monitoring all of its active sockets to **read** data. All sockets with packets pending to send are monitored to **write** data.

The request broker is responsible for the acceptance of a request from any communicating party it is connected to, routing and delivery of this request to the disclosed recipient. When a recipient passes the response packet, the request broker is responsible to deliver the response to sender. Each of the delivery operations always includes a queuing of the packet as the recipient is usually assumed not to be available for the immediate communication.

With each socket connection the request broker associates a priority queue of packets to be delivered to the socket. Priority queues are used to fine-tune system performance and help in preventing some of the race conditions. For example, synchronous packets get higher priorities

than asynchronous ones; remote objects instantiation requests also get higher priority than invocation requests; the context release message broadcast has the lowest possible priority. In addition to aforementioned, to reduce impact on the worker multiplexor execution stack depth packets with responses get a higher priority than packets with requests.

5.2.5. Object Broker

As shortly mentioned above, the object broker implements 3 main functionalities of the distributed objects middleware subsystem: object pool, skeleton and garbage collector. In addition to this, the object broker sends all information about the provided remote object classes (including the stub code) to the naming facility during the startup phase.

In general, the object broker implementation is very similar to the worker, with the only difference that component broker is replaced with object instances pool and instead of calling resource processing methods on components, actual object methods are invoked.

The object broker also implements the Generation Reference Counting (GRC) algorithm as explained in details in section 3.3.4 where possible garbage collection strategies were discussed.

5.2.1. Active XML Parser

The parser component is a subsystem that is responsible recursive Active XML document parsing and passing its active XML elements for processing to the ICC subsystem. First of all it is important to state, that the tile parser is not a part of core system, but is rather a conventional replaceable MOSAIC component of which is a part of *sys.content* package.

Parsing is invoked by the system via calling the asynchronous boomerang event named `content:parse`. Each event instance contains an Active XML document that should be parsed and a callback that must be triggered when parsing of the document is finished. Once a callback is triggered, the parsed document is sent back to the web server and along with response headers that were set during the processing phase, delivered to the client's web browser.

By default, parser operates in breadth-first search (BFS) mode. That means that in the first row first BFS level tiles are processed. In this situation components are receiving tile enclosed DOM document fragment and are able to influence deeper tiles parameters or behavior.

Let's illustrate the usage of the BFS parsing mode on the example below:

```
<tile:if-not xmlns:tile="http://www.bondari.net/mosaic/system-resource/tile" anonymous="1">
Please <a href="/login">click</a> to login!
</tile:if>
<tile:if xmlns:tile="http://www.bondari.net/mosaic/system-resource/tile" anonymous="1">
Hello <tile:user-name />!
</tile:if>
```

It means that if `<tile:if-not />` contains `<tile:user-name />`, first will be parsed the `if-not` tile. Component that processes `if-not` tile will receive also inner DOM nodes of the tile and decide whether it should output it (user is or is not logged in). This allows tiles from **higher** levels to influence processing of tiles in **lower** levels. In the case that `if-not` will not output `user-name` in its processed fragment, or replace `user-name` with something else, `user-name` will never be parsed.

There are also situations where higher level tile processing should be influenced by lower level tiles processing. If this is the case, it is possible to instruct the parser to employ a Depth-first search algorithm on the specific tile node, thus resulting in process of **lower** level tags first. This may be accomplished by setting a parser attribute on the tile to "*DFS*" value. Example below illustrates this situation.

```
<tile:xslt tile:parser="DFS">
	<input>
		<tile:news feed="http://rss.cnn.com/rss/cnn_topstories.rss" />
	</input>
	<stylesheet>
		<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
			<xsl:output method="xml"/>
			<xsl:template match="channel">
				<h1><xsl:value-of select="title" /></h1>
				<xsl:apply-templates select="items/item" />
			</xsl:template>
			<xsl:template match="item">
				<h2><xsl:value-of select="title" /></h2>
				<div><xsl:value-of select="description" /></div>
			</xsl:template>
		</xsl:stylesheet>
	</stylesheet>
</tile:xslt>
```

Parsing and processing of this fragment would result in the application of the XSLT style sheet on the output of the `<tile:news />` tile.

Chapter 6. **Evaluation**

6.1. Example applications

In order to better demonstrate the capabilities and limitations of the MOSAIC framework several functional components and applications were developed and are bundled with the framework source code. Along with general purpose system components such as session, request/response headers, URI manipulation, and the application of XSLT transformations, included are three simple applications: content management system, RSS news reader and e-shop application. Though being small and simple, they were developed in the way that makes it clear how a fully functional application would be developed. All these components and applications are split into the following packages located in the `/src/packages` folder: `sys.utils`, `sys.session`, `cms`, `ecommerce`, `news`, `shipping` and `xslt`.

6.1.1. Client session

The client session application represents a replacement for the PHP built-in session functionality in the MOSAIC environment. The session application contains of 2 components: *stateful component* that is responsible for:

- retrieval and setup of session cookie
- keeping the session cookie value during the content request processing cycle
- providing a convenient interface for accessing the session data for other components
- communicating with session storage component

This component is a part of the *sys.utils* package, which should be added on any node that requires access to the session. The component is registered as a kernel module, thus the instance can be referenced directly via the `$sys->session` global reference. It provides 4 methods that are trying to mimic original PHP session behavior.

`$sys->session->start()` starts the session, reads or creates a session cookie by using the *headers* management component. `$sys->session->getSessionID()` returns current session unique identification value.

`$sys->session->getVar()` (resp. `$sys->session->setVar()`) are used to get (resp. set) data from session storage. These methods are implemented through firing the

synchronous boomerang events `session:get` (resp. `session:set`). These events are received by the session storage component that runs on the particular node.

Session storage component is a *stateless component* that is responsible for keeping session data between content requests, i.e. providing this data amongst its entire executed context. It is subscribed to `session:get` and `session:set` events. These events always contain a session identification string, variable name. `session:get` event is populated with variable value caller wants to get from session. `session:set` event contains variable value caller wants to place into the session.

The session components pair represents an elegant implementation of simplicity and independence with the functional application subsystems. In the future, session storage component may be replaced with a more sophisticated session storage mechanism without the need of reimplementation of any other components including the session access component in particular. The session storage component must only be a subscriber to the events mentioned above.

6.1.2. Headers manipulation

The headers manipulation component was mainly developed in order to provide a cookie placement and reading capability for the session component. The session component and headers manipulation component are part of *sys.utils* package, but opposite of this is a *stateless kernel module* referenced as `$sys->headers`. The component implements all the necessary headers manipulation functionalities such as: getting a specific or all request headers (`$sys->headers->getRequestHeaders()`), adding a response header (`$sys->headers->addResponseHeader()`) and, finally, providing all response headers (`$sys->headers->getResponseHeaders()`). Header data is stored inside the shared context data area (see Figure 13. Execution Context for reference).

6.1.3. URI manipulation

The URI manipulation component provides access to request URI data, (i.e. path and GET request variables). This stateful component is a part of sys.utils package and is accessible as a kernel module *url* through the `$sys->url reference`. This component is initialized with the URI data on the node that processes the initial content request. This data is stored in the shared context area, thus being available on all nodes. To access and manipulate the URI and request variables the data application components should use `$sys->url->getURI()`,

`$sys->url->setURI()`,`$sys->url->getRequestVariable()`, `$sys->url->setRequestVariable()` calls.

For better performance results, each component instance keeps a local copy of the URI related data without need to send a remote request to the shared context data each time some component requests these data. However the *url* component allows the modification of request variables and the URI path during the processing of the request. Should it come to the request URI data modification action, *url* module issues a notification event that is delivered to all other nodes, instructing them to invalidate cache data and reload it upon demand from the shared context pool. This is done in order to synchronize local caches between all nodes.

6.1.4. Content Management System

The content management system (CMS) package's role is to provide the initial document for parsing based on the current request URI. This component hooks on the special initial tile issued by the content generation subsystem called `content` (i.e. `<tile:content xmlns:tile="http://www.bondari.net/mosaic/system-resource/tile" />`). Based on the information received from the URI manipulation module, the CMS module passes content of the file from disk that is located in the same folder as was requested (e.g. request to the page /about triggers the output of a file from the `<packagefolder>/documents/about/document.xml` folder). These files usually contain a mix of XHTML tags and other tiles that will be recursively processed by responsible the MOSAIC components.

The content generation component organization illustrates the supreme flexibility of the MOSAIC based applications in general. It is very easy to replace whole content management system package with another implementation of such a component. All that needs to be done is to assure that the new component outputs content upon content tile processing.

The CMS package also contains another supplementary component that processes `mosaic-version` tile to output the current version of the MOSAIC project and the tile calendar that draws the simple HTML calendar control. Usage of this control will be illustrated later on in section Generic RSS news reader where this snipped will be integrated with news reader application to display RSS items only for a given date a user clicks on.

6.1.5. XSLT transformation component

The XSLT transformation component is unique. Unlike the components mentioned above, that are accessible through the kernel module interface or events notification mechanism, the XSLT transformation component demonstrates usage of *tile* inter component communication paradigm. This component provides a tile named `xslt`, which is to be used to perform XSLT transformations during the document parsing phase. In order to perform the transformation, the component should output following XML fragment.

```
<tile:xslt xmlns:tile="http://www.bondari.net/mosaic/system-resource/tile">
	<input>
		…. Input XML document …
	</input>
	<stylesheet>
		<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
			… XSLT stylesheet …
		</xsl:stylesheet>
	</stylesheet>
</tile:xslt>
```

This tile will be processed by the XSLT transformation component and be replaced with the transformation result in a same manner as illustrated on the Figure 5. Active XML processing example).

6.1.6. Generic RSS news reader

The RSS news reader is the first application that illustrates the usage of high level components in order to generate web page content. The news package comprises of a single stateless component that implements a tile named news. This component is able to read an RSS feed provided as a tile attribute, format it in different manners and also filter the items by the date selected by the user. Tile news takes the following attributes: *@feed*, *@limit*, *@calendar*, *@verbose*, *@output*.

The attribute *@feed* contains a RSS feed URL to load. Feeds are loaded only once. Feed items are stored in the component instance. Such pre-caching of items is possible due to the fact that news component is declared as stateless, thus all tile processing requests are performed by the same singleton component instance.

The attribute *@limit* instructs component to serve only last N items specified in this attribute.

The attribute *@verbose* controls the formatting stylesheet, whether component needs to use verbose stylesheet or short format stylesheet.

The attribute *@output* may be set to "*xml*" value in order to force component not to apply any style sheets and output raw XML data.

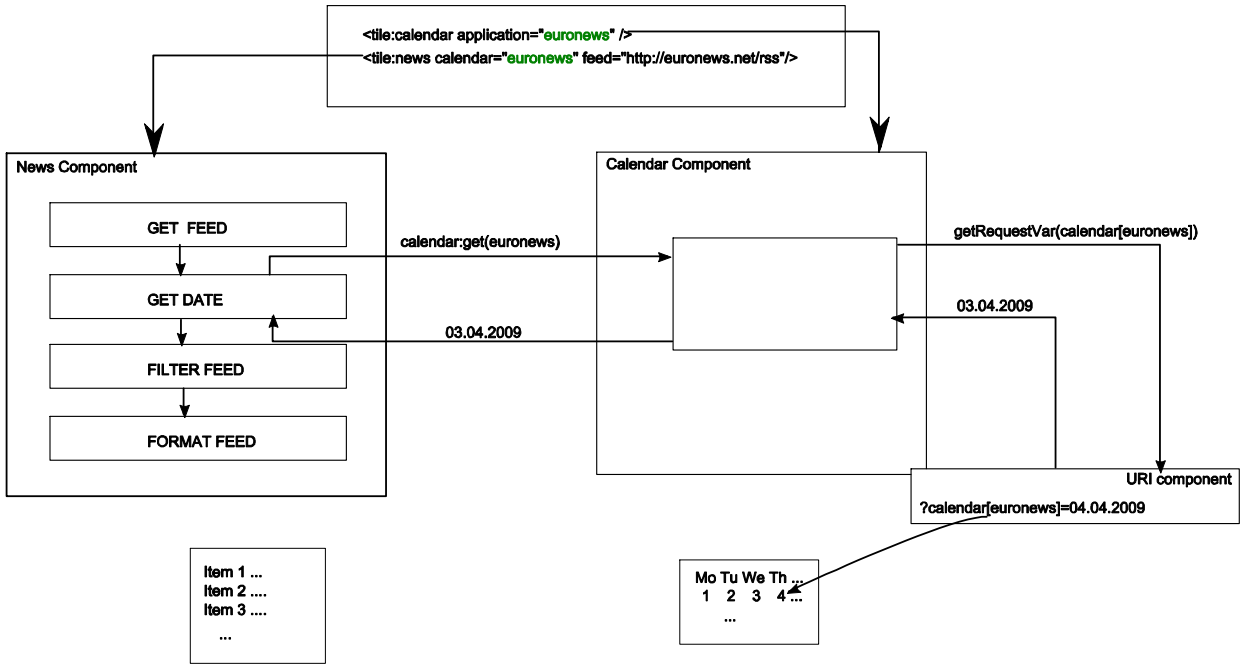

Figure 14. Calendar component integration

The attribute *@calendar* tells the component which calendar instance is used for date filtering.

The *calendar* tile has an *@application* attribute, that uniquely identifies for which application this calendar tile is drawn (e.g. a news module). In order to get the currently selected date, the application component issues a synchronous boomerang event *calendar:get*, with the only parameter application. The calendar component contacts the URI module to get the current request variables and sends back the date user clicked on (Figure 14. Calendar component integration).

6.1.7. E-Shop

The e-shop application implements a basic electronic commerce application. Its primary goal is to illustrate the usage of distributed objects and session components. As mentioned in 5.2.3 distributed objects are used by the system core in order to provide shared context data access, though this application demonstrates higher level employment of this subsystem.

The application comprises of 4 components: *engine*, *products*, *cart* and *shipping calculator*. In order to be able to split the e-shop between 2 nodes and thus to illustrate passing-by-reference object procedure, shipping component is organized into the separate package.

The engine component is the smallest component. It implements a single tile called `shop`. With the help of a synchronous boomerang event the engine reads all available product ids from the *product* component and then sequentially outputs the tile product for every product id it received. The tile `product` is handled by the product component and is used for a formatted output of the product information. The products component is subscribed to both the tile `product` and the event *ecs:get-product-ids*. Integration of the *engine* and *products* component is also illustrated on the Figure 15. E-shop engine and products integration

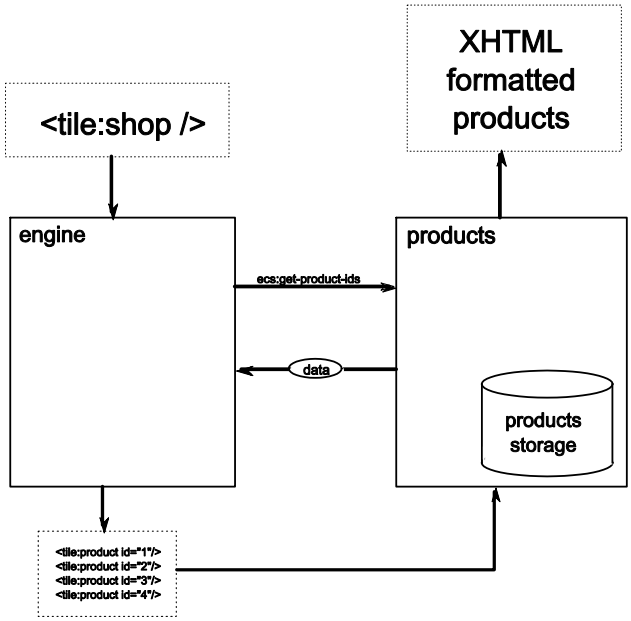

Figure 15. E-shop engine and products integration

Cart component is a bit more interesting. In addition to the tile `cart` that is used to output the current cart contents and an event *ecs:get-active-cart* that is to be triggered in order to get the active cart object instance, the cart component provides a distributed object `Cart`. The cart holds information about products, quantities and shipping cost information for the current session. In-between of page reloads, the `Cart` reference is stored within the session component.

The `Cart` implements a minimal required interface that a shopping cart might have: `addItem()`, `setQuantity()` and `getSubTotal()`.

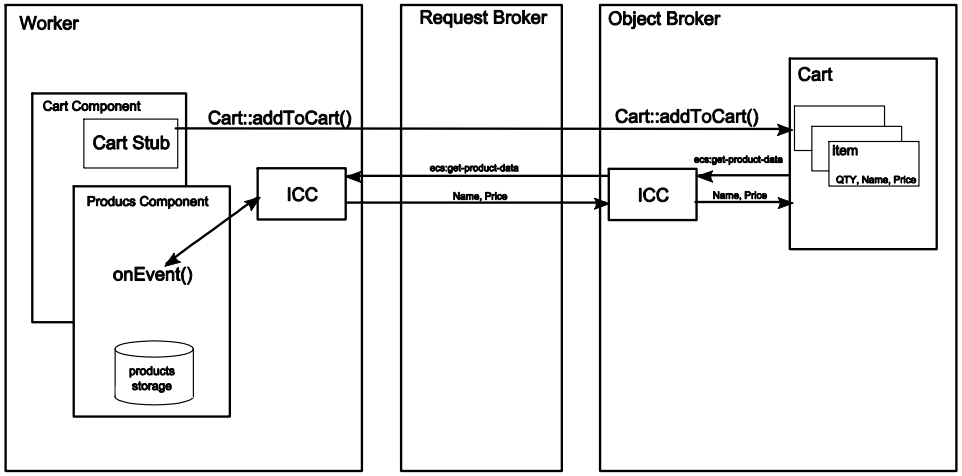

Figure 16. ICC access from object broker

The function `Cart::addItem($productId, $quantity)` is interesting by itself as it is using the ICC subsystem to contact the *Products* component in order to fetch product data such as the price and name for the given product ID. Though object broker is a separate node process, access to ICC resources from remote object code is still completely transparent as they were accessed from a component code (Figure 16. ICC access from object broker). Other `Cart` object methods are straightforward and can be seen in the source code of the demo application supplied.

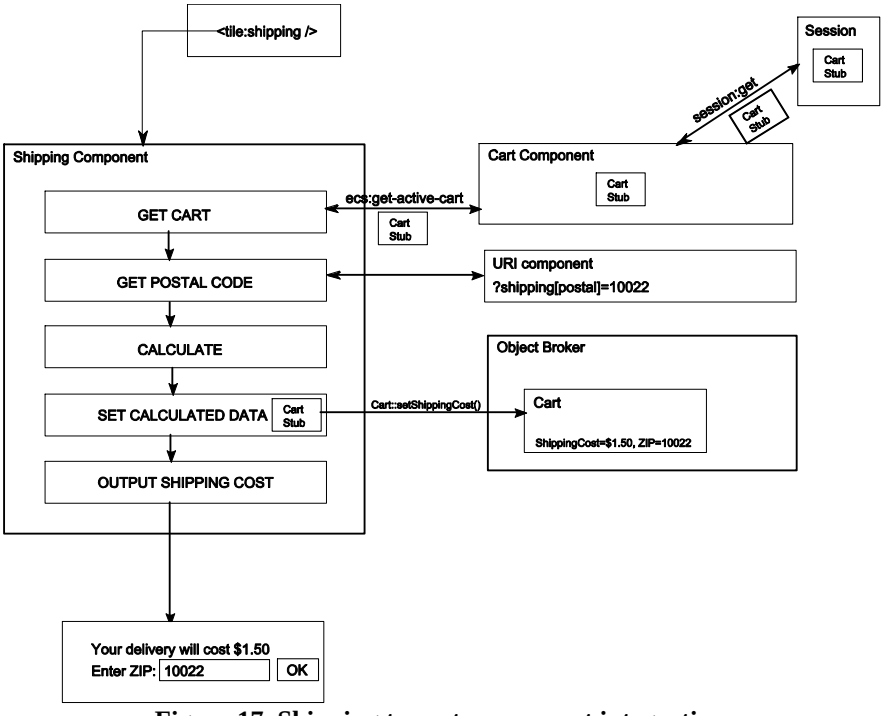

Figure 17. Shipping to cart component integration

The shipping component provides a `shipping` tile that outputs a simple form where the user can enter his address information (5 digit US postal code) and see the delivery cost for his location. When a zip code is submitted, the shipping module triggers the `ecs:get-active-cart` event notification and receives a `Cart` object stub instance (i.e.

`Cart` by reference). It calculates the shipping prices and puts it along with address information to the cart. From this moment forward, the Cart instance contains this additional shipping data that may be accessed by other components. To demonstrate, this cart component will also display the shipping cost during the next page load.

6.2. Benchmarks

6.2.1. Configuration

Hardware

All benchmarks where performed on 3 physical computers connected to a 100 megabit Ethernet local network. Average network latency was measured to be 1.5±0.5 ms. For the sake of deployment simplicity all MOSAIC instances where running under VMware virtual machines.

Computer A configuration: Intel Core Duo 2 processor 2.13 GHz with virtualization support, single virtual machine instance, 2 logical processors.

Computer B configuration: Intel Core Quad processor 2.40 GHz with virtualization support, 2 virtual machines, 2 logical processors each.

Computer C configuration: Intel Core Duo processor 1.83 GHz with virtualization support, 1 virtual machine, 2 logical processors.

All configurations had been allocated sufficient RAM, and the swap partition was disabled.

Software

Operating system is Linux, Ubuntu 8.04 LTS distribution, kernel 2.6.24-23-server SMP, PHP version 5.3.0 beta 1, Apache version 2.2.8 with libapache2-mod-scgi 1.12-0.2 module enabled. For benchmarking ApacheBench software version 2.0.40-dev was used.

Test methodology

Every application was tested in two configurations: a single node configuration where components were communicating only through Unix sockets IPC and a distributed configuration where the components are distributed on multiple nodes. Tests were performed with concurrency levels 1, 2, 3, 5, 10, 20, 30, 40 (i.e. 1, 2, 3, 5, 10, 20, 30 and 40 simultaneous users simulation). On the numbers higher that mentioned above, system performance was not changing either direction.

6.2.2. News application

Packages used: content management system, RDF news reader, XSLT transformation component. The page consists of 2 news tiles that display the full contents of 2 different RSS feeds, each of them transformed by the XSLT transformation style sheet.

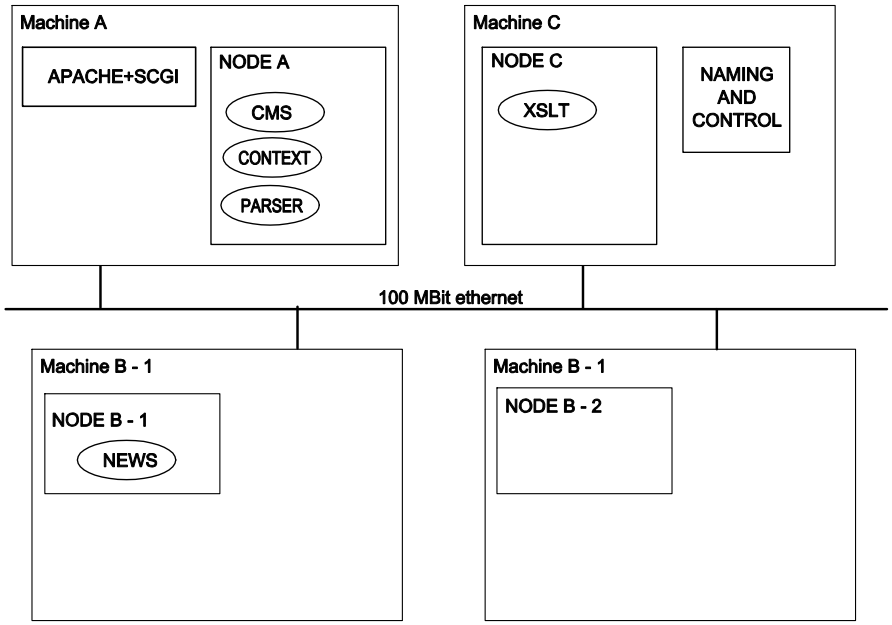

Figure 18. Distributed news application configuration.

Concurrency Level	Single Node, req/s	Distributed, req/s
1	21,25	2,38
2	21,04	4,35
3	21,30	5,59
5	21,35	7,75
10	21,32	11,47
20	21,31	14,48
30	21,13	15,12
40	20,89	14,59

Table 2. News application benchmarking results

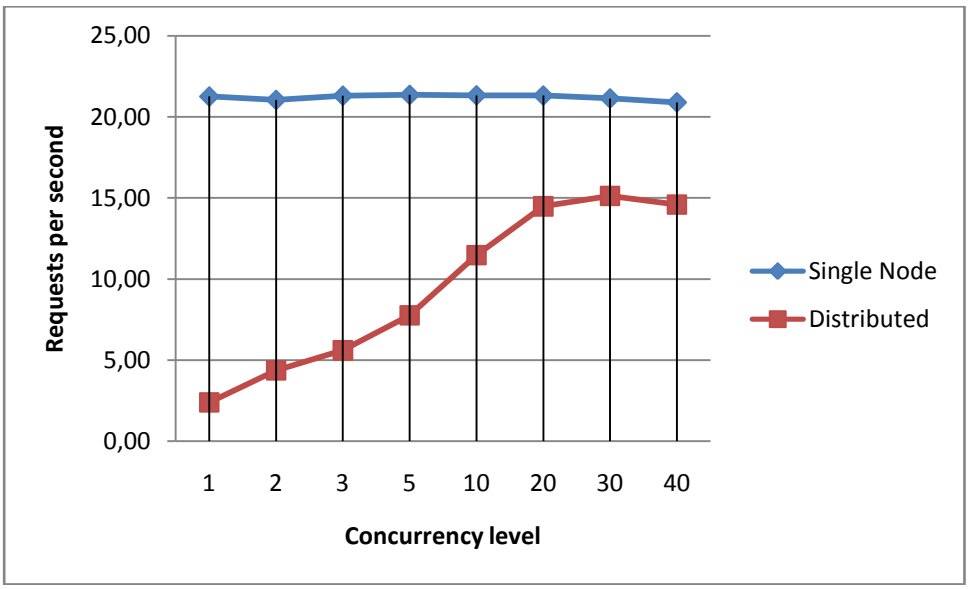

Figure 19. News application benchmarking results diagram

6.2.3. Complex e-shop+news application

The e-shop application test case was designed to be the most network communication intensive. During a single page load, the application adds all the items into the cart one by one, calls a shipping price calculation routine, displaying all products list, shipping price information and the cart contents. The cart is also stored to the session component. The shop page also includes 2 news feeds listed on a side bar of the page, outputting last 3 items of each feed.

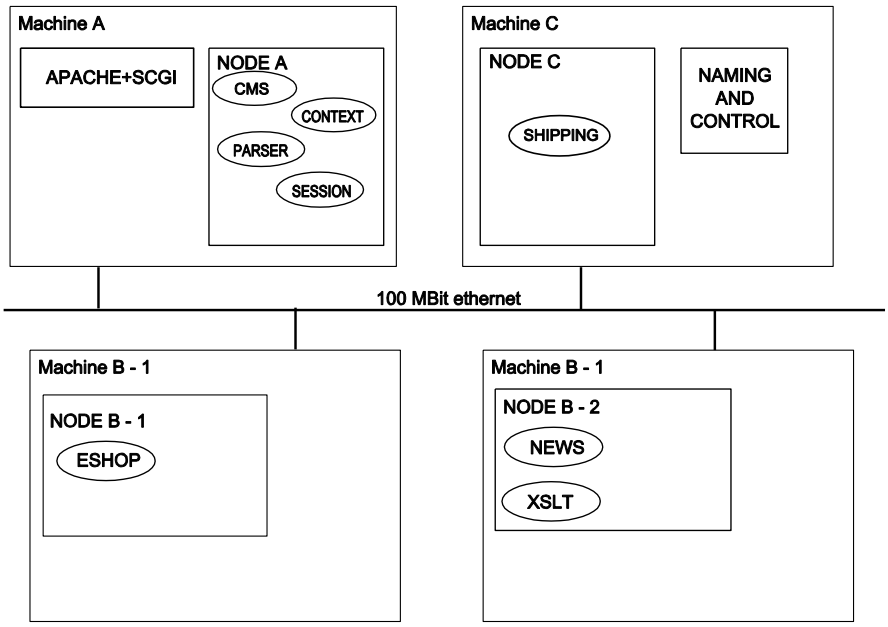

Figure 20. Complex eshop + news application configuration

Concurrency Level	Single Node	Distributed
1	19,11	1,21
2	19,00	2,38
3	19,17	2,55
5	19,33	2,94
10	18,83	3,23
20	18,20	3,56
30	17,79	3,90
40	17,75	3,44

Table 3. E-shop + news application benchmarking results

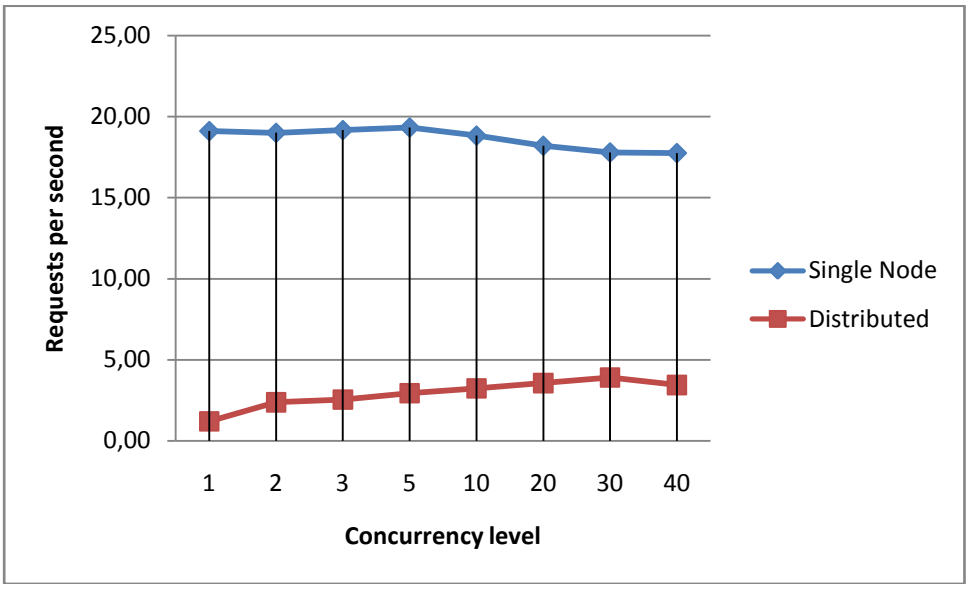

Figure 21. E-shop + news application benchmarking results diagram

6.2.4. Distributed CPU load application

This test case consists of 4 tiles that are served by 4 dummy components that intensively use the CPU time.

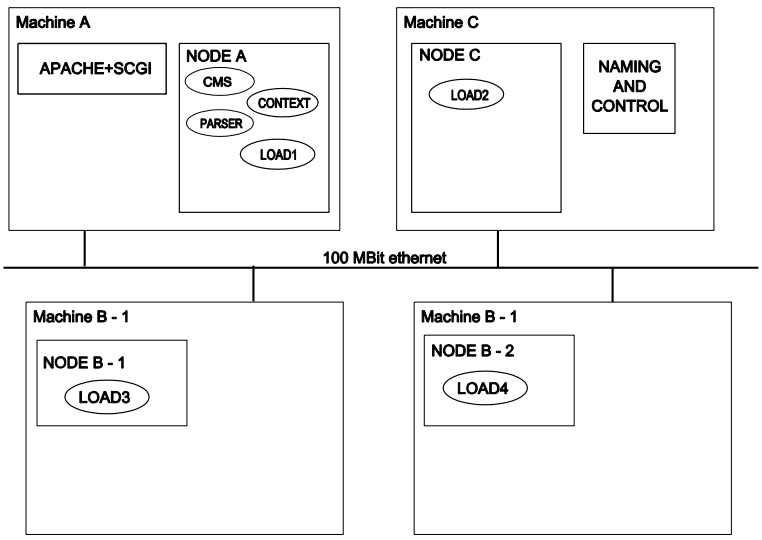

Figure 22. CPU load application benchmark configuration

Concurrency Level	Single Node	Distributed
1	1,76	2,82
2	1,76	4,01
3	1,76	4,75
5	1,77	4,85
10	1,73	5,10
20	1,72	5,38
30	1,71	5,56
40	1,67	5,61

Table 4. CPU load application benchmark results

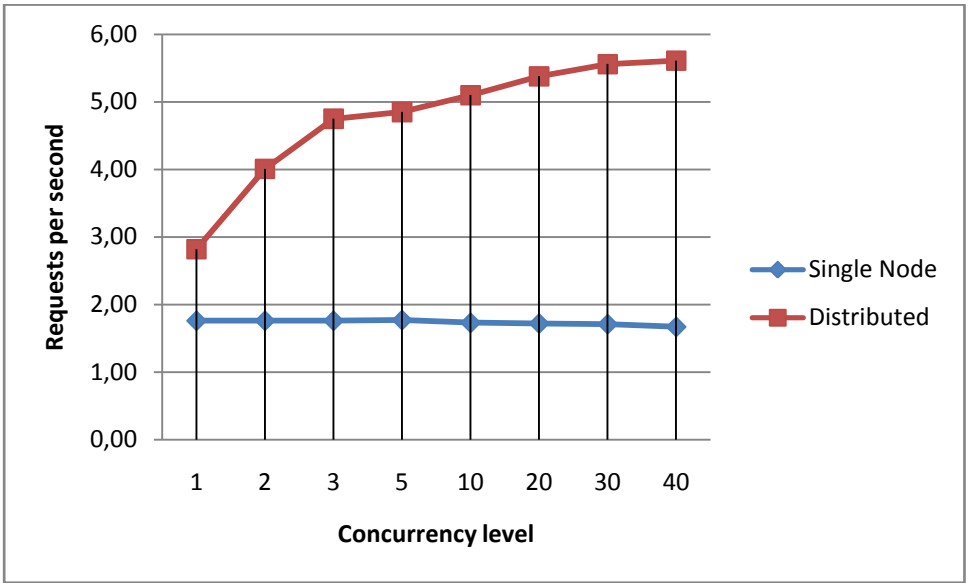

Figure 23. CPU load application benchmark results diagram

6.3. Evaluation

6.3.1. Application development

During the development of the demo application the MOSAIC framework has proven to be an easy-to-use and logical tool. Usage of the ICC primitives, such as events and tiles, were guiding me as an application developer to design independent, replaceable and self-focused application components and then assembling these components in the application scope. I did not encounter any lack of traditional procedural RPC programming when using the implementation of DEBS paradigm.

The remote objects functionality had also provided a completely transparent and convenient way to access the same object instance from different nodes of a site and passing of such objects by reference.

During the testing I was also benefiting from a migration transparency of application components. To move a component from one node to another turned out to be as simple as to add a single line with the path to the component code to a node configuration file, comment out the line on the node I was moving the component from and restart *just* these two nodes. The site was reconfigured automatically and again being completely functional in a matter of seconds.

6.3.2. Benchmark results

As explained in the previous section, benchmarking was performed on 3 different configurations, each representing a different class of applications in a possible real world usage of the framework.

The News application represents a smaller application with an average-to-low demand for server CPU time, memory and ICC communication. During the page buildup MOSAIC processed a total of 8 tiles (content tile, left navigation, top navigation, 2 news RDF feeds + 2 XSLT transformation tiles, and the system version tile on the bottom of the page). The processing of only 2 XSLT tiles was creating a higher CPU load.

The single node installation benchmark had shown stable application performance during the increase of the concurrency level. The operating system resources monitoring also did not show any increase in CPU or memory usage when increasing number of concurrent users. This is an expected result to get on any server application that uses multiplexing mechanisms to process incoming requests.

The multiple node benchmark results had shown increasing performance when the number of concurrent users increased. It is a bit uncommon, though predicable result. Adding extensive network communication to the processing of requests could not add any performance. On the other hand, with more simultaneous request coming, MOSAIC was allocating components to process pending sub-requests while waiting for responses from other nodes having bigger response latencies than a single node installation. This benchmark test perfectly illustrated the benefits of the multiplexed multitasking design of MOSAIC.

The **Complex E-shop and News application** displays a completely different picture. This application imposes a great load on the ICC subsystem and parsing, thus adding a real network latency time had a huge impact on general application performance. During a test case the page load application would process 15 tiles (content, left and top navigation, e-shop catalog tile with 4 product tiles, cart tile, shipping calculator tile, 2 news tiles resulting in 2 XSLT tiles, version tile). Moreover this application intensively uses ICC event primitives (to access

products data, pass Cart object instance etc.) and remote object invocations (to add products, set shipping cost related data etc.)

While a single node benchmark displayed a slight performance decrease comparing to the previous test case, the distributed benchmark displayed a major decrease in performance. Even addition of more simultaneous users did not improve the situation. Although, comparing to a very high CPU load during the single node test (over 80% usage), neither node was showing more than 5%-10% CPU load during the distributed configuration benchmarking. This means that the performance bottleneck was likely networking latency.

Distributed CPU load test was created to illustrate a processing distribution in an application that imposes great demand on computation resources. Based on the test results, in applications where network latency does not play a substantial role, the document processing time is scaled on the number of nodes. This test shows that the MOSAIC system processes tiles true parallel fashion and illustrates the advantages of a multiplexed multitasking system like MOSAIC.

Chapter 7. **Conclusion**

7.1. Conclusion

After evaluating the framework I can proclaim that all goals stated in 1.2 were successfully accomplished. I have managed to create an application framework that provides a simple and convenient infrastructure to create distributable component based web applications purely in the PHP programming language. This infrastructure includes a component model, distributed objects, remote exceptions, inter component communication primitives and other functions that offer a transparent distributed environment to develop such applications.

Usage of an event driven distributed systems paradigm and distributed document paradigm in conjunction with multiplexed data processing was made possible to address and resolve various limitations of PHP development platform and to provide an effective way to perform parallel and distributed execution of the application code on multiple system nodes.

Same paradigms that resulted in the introduction of the *tile* and the *event* inter component communication primitives, in cooperation with the framework's component model, required the developer to create highly reusable, self-focused, independent, easily replaceable and autonomous application components.

Benchmarking of the demo applications revealed that there is room for improvement in terms of performance and scalability of the framework based on different types of applications. These issues are not critical to the goals of the current project and will be addressed in future project development.

7.2. Project's future

The closest goal in the project evolution is to address poor scalability revealed during benchmarking in network intensive applications. In order to achieve the best results, this task should be approached from two different angles. First, it is necessary to decentralize the parser component, thus minimizing network communication between nodes during the tile parsing and processing. Second, framework should be enhanced in a way that it would allow having multiple workers inside a node to increase the level of parallelism on a single machine[17]. This can easily be accomplished as the framework is nearly prepared for this: the object broker process is separated and all communication packets are tagged with the context identification

[17] During the benchmark of the distributed configuration of application average CPU load was quite low comparing to the single node configuration. This indicates a communication bottleneck.

string, thus the request broker will have to implement some kind of context affinity when delivering requests to multiple workers.

The second task for the project future is to implement heterogeneity in terms of programming language, (i.e. add support for remote objects and components programmed in Java). With some limitations this should not be a difficult task as PHP provides a Java integration subsystem that allows the creation of Java objects inside the PHP environment. A high level of MOSAIC components autonomy and independence, the fully object oriented approach and the simplicity of a MOSAIC component interface should render this task almost trivial.

A much more challenging task will be to add replication functionality into the framework, (i.e. having same components running on different nodes, thus adding a high level of scalability and reliability to the site). Upon future demands, MOSAIC may be also extended with security and distributed transactions middleware services.

Bibliography

[1] **Andrew S. Tanenbaum, Maarten Van Steen.** *Distributed Systems: Principles and Paradigm.* 2002.
[2] **Muhl G., Fiege L., Pietzuch P.** *Distributed event-based systems.* s.l. : Springer, 2006.
[3] **X.200.** *Information technology - Open Systems Interconnection - Basic Reference Model.*
[4] **Information Sciences Institute .** RFC793: Transmission control protocol. [Online] September 1981. http://tools.ietf.org/rfc/rfc793.txt.
[5] **Postel, J.** RFC768: User Datagram Protocol. [Online] August 28, 1980. http://tools.ietf.org/rfc/rfc768.txt.
[6] **Goldberg, Benjamin.** *Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme.* New York : s.n., 1989.
[7] **Plainfosse D., Shapiro, M.** *A Survey of Distributed Garbage Collection Techniques.* s.l. : Berlin: Springer-Verlag, 1995.
[8] **DCMI.** Dublin Core Metadata Element Set, Version 1.1. *The Dublin Core Metadata Initiative.* [Online] 01 14, 2008. http://dublincore.org/documents/2008/01/14/dces/.
[9] **W3C World Wide Web Consortium.** URIs, URLs, and URNs: Clarifications and Recommendations 1.0. [Online] September 21, 2001. http://www.w3.org/TR/uri-clarification/.
[10] —. Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online] 11 26, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.
[11] **Open Market, Inc.** FastCGI: A High-Performance Web Server Interface . [Online] April 1996. http://www.fastcgi.com/drupal/node/6?q=node/15.
[12] **Schemenauer, Neil.** SCGI: A Simple Common Gateway Interface alternative. [Online] June 23, 2008. http://python.ca/scgi/protocol.txt.
[13] **Object Management Group (OMG).** *CORBA event service specification version 1.0.* 2000.
[14] —. *CORBA event service specification version 1.1.* 2004.
[15] **Thomas, Anne.** *Enterprise JavaBeans Server Component Model for Java.* 1997.
[16] **S., Vinoski.** *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.* 1997.
[17] **W3C World Wide Web Consortium.** Resource Description Framework (RDF) W3C specification. [Online] http://www.w3.org/RDF/.
[18] *Network Multitasking: Event based machines.* **Kudinov, Pavel.** Moscow : ProfyClub, 2007. HighLoad Conference 2007.
[19] **W3C World Wide Web Consortium.** XSL Transformations (XSLT) Version 1.0. [Online] November 1999, 16. http://www.w3.org/TR/xslt.
[20] **Canonical Ltd.** Ubuntu Linux official page. [Online] http://www.ubuntu.com/.
[21] **The Apache Software Foundation.** *The Apache HTTP Server Project.* [Online] http://httpd.apache.org/.
[22] **CollabNet, Inc.** *Subversion official homepage.* [Online] http://subversion.tigris.org/.
[23] **SUN Microsystems Inc.** *Java Technology.* [Online] http://www.sun.com/java/.
[24] **The PHP Group.** *PHP: Hypertext Preprocessor.* [Online] http://php.net/.
[25] —. PHP 5.3 upgrading guide. [Online] http://wiki.php.net/doc/scratchpad/upgrade/53.
[26] **W3C: Wold Wide Web Consortium.** Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online] November 26, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.
[27] *RAP - RDF API for PHP.* [Online] http://www4.wiwiss.fu-berlin.de/bizer/rdfapi/index.html.

Appendix A: Contents of the attached DVD

Attached DVD contains complete MOSAIC sources with 3 startup configurations to start the same demo application on 1, 2 and 4 nodes. The disk also contains complete code documentation in HTML format, the installation instructions for Linux operating system and a VMware image of the configured Ubuntu Linux 8.04 LTS with all necessary software installed and set up to start MOSAIC demo applications. The master thesis paper in PDF format is also included into this DVD.

DVD Folders Map

```
/installation
    readme.txt  (installation instructions on Linux)
/reference
    index.html  (code documentation)
/sources  (complete MOSAIC sources)
    /bin
        /1node  (single node application startup scripts)
            run  (BASH script to run application)
            node1.php  (Node bootstrap)
            node1.ini  (Node configuration)
            ns.php  (Naming and control agent bootstrap)
            ns.ini  (Naming and control agent configuration)
        /2nodes  (2 nodes application startup scripts)
            run
            node1.php
            …
        /4nodes  (4 nodes application startup scripts)
            run
            node1.php
            …
    /config  (configuration files examples)
    /debug  (MOSAIC logs file viewer for Firefox web browser)
    /src
        /lib  (Shared libraries)
        /ns  (Naming and control agent code)
        /packages  (Application components)
        /system  (Core system code)
/thesis
    thesis.pdf  (Master thesis paper)
/vm  (Configured virtual machine to start MOSAIC application)
/vmplayer  (VMware VM Player v2.5.2 installation packages for Linux and Microsoft Windows)
```

Appendix B: Installation instructions

Installation

This chapter describes installation process on Ubuntu Linux of all components required by MOSAIC. Process of installation on other UNIX based operating systems should be very similar.

MOSAIC framework requires PHP 5.3 CLI and any SCGI aware web server, such as Apache2, NGINX (http://www.nginx.org) or LightHTTPD (http://www.lighttpd.net).

To install and run MOSAIC framework demonstration applications do the following steps:

1. Prepare Ubuntu Linux by installing all necessary software

```
sudo apt-get install wget apache2 subversion libapache2-mod-scgi
gcc binutils-doc autoconf automake1.9 bison flex make libc6-dev
libxml2 xml-core libxml2-dev libcurl3 libcurl4-openssl-dev
libmcrypt-dev libsnmp-dev libtidy-dev libxslt1-dev
```

2. Get the latest PHP 5.3 version from http://www.php.net PHP 5.3 RC 1 is downloadable from http://downloads.php.net/johannes/php-5.3.0RC1.tar.gz

```
 wget http://downloads.php.net/johannes/php-5.3.0RC1.tar.gz
```

3. Unpack and compile PHP using following command

```
./configure --enable-pcntl \
     --with-curl \
     --with-curlwrappers  \
     --enable-mbstring  \
     --with-mcrypt \
     --with-mhash  \
     --enable-shmop \
     --with-snmp \
     --enable-soap \
     --enable-sockets \
     --enable-sqlite-utf8 \
     --enable-sysvmsg \
     --enable-sysvsem \
     --enable-sysvshm \
     --with-tidy \
     --enable-wddx \
     --with-xmlrpc \
     --enable-zip \
     --with-xsl && make && sudo make install
```

4. Checkout source code of MOSAIC framework. Anonymous read-only access is allowed.

```
svn checkout svn://dev.bondari.net/mosaic/trunk mosaic
```

5. Create an APACHE configuration file "/etc/apache2/sites-available/mosaic".

```
<VirtualHost *>
    ServerAdmin support@example.com
    ServerName mosaic
    SCGIMount / 127.0.0.1:9001
    DocumentRoot /home/mosaic/mosaic
    <LocationMatch "^/doc/*">
        SCGIHandler Off
    </LocationMatch>
</VirtualHost>
```

6. Enable SCGI Apache module and mosaic site, restart apache.

```
a2enmod scgi && a2ensite mosaic && /etc/init.d/apache2 force-
reload
```

7. Start demo MOSAIC application by executing the <mosaicfolder>/bin/1node/run

```
/home/mosaic/mosaic/bin/1node/run
```

8. Open MOSAIC in your browser by opening the page in browser (e.g. http://192.168.1.108/)

Using the supplied VMware image

For convenience, supplied DVD contains VMware image with a pre-installed and pre-configured according to the steps above. To login, please use the user name "`mosaic`" with the password "`mosaic`". The network is configured in DHCP mode. In case if virtual machine does not see the **eth0** device just remove `/etc/udev/rules.d/70-persistent-net.rules` and restart the virtual machine.

The MOSAIC framework source code is located in the `/home/mosaic/mosaic` folder. To start MOSAIC test applications in 1, 2 or 4 nodes configuration, start the `./run` script in the `/home/mosaic/mosaic/bin/1node`, `/home/mosaic/mosaic/bin/2nodes` or `/home/mosaic/mosaic/bin/4nodes` folders. Then open the http://Server_IP_Address/ in any browser to start a MOSAIC demo web application.